UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Telecomunicació
i Aeroespacial de Castelldefels

BACHELOR DEGREE THESIS

TFG TITLE: A new Deep Reinforcement Learning architecture for autonomous UAVs

DEGREE: Grau en Enginyeria Telemàtica

AUTHORS: Guillem Muñoz Ferran

ADVISOR: Cristina Barrado Muxí

DATE: 31 d'agost de 2018

Title : A new Deep Reinforcement Learning architecture for autonomous UAVs

Authors: Guillem Muñoz Ferran

Advisor: Cristina Barrado Muxí

Date: August 31, 2018

Overview

Recent improvements in computation and algorithmic research, together with the rising era of Big Data, have allowed Artificial Intelligence increase its popularity within masses. The recent publication of the Deep Q-Network (DQN) algorithm, which combines Q-learning with deep neural networks, has been demonstrated as being able to learn how to solve complex task, such as playing Atari games, in an unknown environment solely by gathering experience. These conditions open the door for many other applications, such as autonomous vehicles, doctors or production chains. Moreover, the preceding work of this project was focused on building a baseline architecture for enabling Unmanned Aerial Vehicles (UAVs) learn how to behave autonomously.

In this project we provide different architectures for scaling this solution. To evaluate the convergence of the algorithm, we create challenging tasks concerning obstacle avoidance and goal position reaching inside a realistic simulated environment. The provided solution allows UAVs to autonomously move in three dimensions as well as controlling and modifying their velocities. Modifications in the architecture provide different approaches for learning, which are evaluated together with its training efficiency metrics and testing results.

The development has been focused on integrating Deep Learning and Reinforcement Learning tools such as Keras and OpenAI Gym in order to build a modular and accessible framework capable of training and testing DRL models for autonomous UAVs within simulated environments. Results of the carried experiments show multiple enhancements compared to previous research and work, along with providing useful insights for potentially identified improvements. In this project, we have been able to successfully beat the existent baseline Double Deep Q-Learning architecture for autonomous UAVs, obtaining a 49% more of average reward and no collisions, on a non-trivial task within a realistic simulated environment.

Título: A new Deep Reinforcement Learning architecture for autonomous UAVs

Autores: Guillem Muñoz Ferran

Director: Cristina Barrado Muxí

Fecha: 31 d'agost de 2018

Resumen

Millores recents en les àrees d'investigació de la computació i l'algorítmica, juntament amb la creixent era del Big Data, han permès a l'Intel·ligència Artificial guanyar popularitat entre les masses. La recent publicació de l'algorisme Deep Q-Network (DQN), que combina Q-learning amb xarxes neuronals profundes, ha demostrat ser capaç d'aprendre a resoldre tasques complexes, com jugar a jocs d'Atari, en un entorn desconegut i només a través de l'experiència. Aquestes condicions obren la porta per a moltes altres aplicacions com vehicles, doctors o cadenes de producció autònomes. La tesi prèvia a aquest projecte va estar focalitzada en construir una arquitectura base, habilitant vehicles aeris no tripulats (UAVs) a aprendre com comportar-se de manera autònoma.

En aquest projecte proveïm aquella solució de diferents arquitectures per tal d'escalar-la. Per a evaluar la convergència de l'algorisme, creem tasques desafiants consistent en evitar obstacles i en arribar a una destinació, dins d'un entorn de simulació realista. Aquesta solució permet a UAVs moure's de manera autònoma en tres dimensions, a més de poder variar i controlar la seva velocitat. Les modificacions de l'arquitectura proporcionen diferents enfocaments per a aprendre. Tots són avaluats amb l'eficiència de les mètriques d'entrenament i els resultats satisfactoris de les proves.

El desenvolupament ha estat centrat en integrar eines de Deep Learning i Reinforcement Learning com Keras o OpenAI Gym amb la finalitat de construir un framework modular i accessible capaç d'entrenar i provar models de DRL per a UAVs autònoms en entorns simulats. Els resultats dels experiments portats a terme mostren múltiples millores en comparació amb el treball i la investigació prèvia, proporcionant a més idees útils per a millores potencials identificades. En aquest projecte hem aconseguit superar amb èxit l'arquitectura de Double Deep Q-Learning anterior per a UAVs autònoms, obtenint un 49% més de recompensa mitjana i sense cap col·lisió, en una tasca no trivial i dins d'un entorn de simulació realista.

Acknowledgements

Years ago back in my childhood, I remember falling in love with maths. My passion for engineering and computer science made me start this degree in networks and telecommunications. Almost a year ago, motivation brought me to study machine learning. But the key factor in this thesis has been inspiration. Cristina Barrado, my supervisor, has inspired me in working hard for what you love and what you aim, leaving aside other external factors.

I would like to thank also people who supported me during this whole journey, including my family, friends and girlfriend. Deep inside myself I know one day I will end up pursuing a doctorate, but until now, this thesis describes somehow my small piece of contribution to science and research.

Declaration

This thesis has been totally developed, conducted and written by myself. All external sources both used or applied have been carefully declared in the text with a formal reference. This thesis is only used for the *Grau en Enginyeria Telemàtica* carried at *Escola d'Enginyeria de Telecomunicacions i Aeroespacial de Castelldefels* (EETAC) belonged to the *Universitat Politècnica de Catalunya* (UPC).

Guillem Muñoz
August 2018

Contents

List of Figures

List of Tables

Nomenclature

$\mathcal{A}(s)$	set of all possible actions in state s – action space
$\mathcal{S}$	set of all states – state space
a	action
r	reward
s, s'	state
A_t	action at time step t
G_t	return (cumulative discounted reward) following time step t
R_t	reward at time step t to S_{t-1} and A_{t-1}
S_t	state at time step t
T	final time step of an episode – terminal step
t	discrete time step
μ	behaviour policy used to select actions while estimating values for policy π
π	policy, decision-maker
π_*	optimal policy
δ_t	temporal-difference error at t
θ, θ_t	vector of weights underlying an approximate value function
θ^-	vector of weights of the target network $\hat{Q}$
Q, Q_t	array estimates of action-value function q_π or q_*
$q_*(s,a)$	value of taking action a in state s under the optimal policy π_*
$q_\pi(s,a)$	value of taking action a in state s under policy π
V, V_t	array estimates of state-value function v_π or v_*
$v_*(s)$	value of state s under the optimal policy π_*
$v_\pi(s)$	value of state s under policy π (expected return)
y_t	learning target

$q(s,a;\theta)$ approximate value of state-action pair s,a given weight vector θ
$v(s;\theta)$ approximate value of state s given weight vector θ

α learning rate or step-size parameter
ε probability of taking a random action in an ε-greedy policy
γ discount factor
τ target factor

$\text{argmax}_a f(a)$ a value of a at which $f(a)$ takes its maximal value
$\doteq$ an equality relationship that is true by definition
$\mathbb{E}[X]$ expectation of random variable X
$Pr\{X = x\}$ probability that the random variable X takes on the value x

ϕ agent's relative heading to the goal – track
ψ yaw angle relative to initial orientation
d_x agents x distance with goal
d_y agents y distance with goal
d_z agents z distance with goal
dg_x agents distance with x geofencing
dg_y agents distance with y geofencing
dg_z agents distance with z geofencing
p_x agents global x position
p_y agents global y position
p_z agents global z position
v_x agents x velocity
v_y agents y velocity
v_z agents z velocity

AGI Artificial General Intelligence
AI Artificial Intelligence
API Application Programming Interface
CNN Convolutional Neural Network
DDQN Double Deep Q-Network
DQN Deep Q-Network
DRL Deep Reinforcement Learning
JNN Joint Neural Network

JQN Joint Q-Network
MC Monte Carlo
MDP Markov Decision Process
RL Reinforcement Learning
RPC Remote Procedure Call
TD Temporal Difference
UAS Unmanned Aerial System
UAV Unmanned Aerial Vehicle

Chapter 1

Introduction

1.1 Artificial intelligence

Artificial intelligence (AI) is a field of computer science that aims to enabling computer systems encapsulate cognitive functions. This involves including adaptive and learning capabilities which lead the systems to a self-improved state. In computer science, AI research is defined as the study of *intelligent agents* any system that perceives its environment and takes actions that maximize its chance of successfully achieving its goals.

When looking into new paradigms, in [Vin93] Vernor Vinge explains the concept of Singularity, the possible causes and how humans may survive after it. The Singularity involves a self evolving intelligence which can improve more rapidly than one could ever imagine. This artificial super intelligence possesses more cognitive capabilities than gifted to humans beings. Back to 1993, Vinge already stated:

"Within thirty years, we will have the technological means to create superhuman intelligence. Shortly after, the human era will be ended."

A similar approach of what Nick Bostrom shows in [Bos14], where he suggests that new super intelligence could replace humans as the dominant lifeform, besides defining super intelligence as:

"an intellect that is much smarter than the best human brains in practically every field, including scientific creativity, general wisdom and social skills."

Experts in the field remark that the first super intelligent machine will be the last invention that human may ever need to make, but that there is a long road ahead in order to potentially reach this point. Recent research advances include the distinction between two main types of artificial intelligence.

Artificial General Intelligence (AGI) predicted to happen around 2040 by [RKT18], when an intelligence which can be as intelligent as human beings would be ready. Experts

explain that there is not a predefined definition of it and that it should be able to learn, represent knowledge, plan, take decisions under uncertainty, communicate in a natural language and use these skills towards common goals to be an AI-complete machine. The biggest constraint here is being able to figure out how the human brain, a 20 W machine product of millions of years of evolution, actually works. So, even if we are able to find out how a single neuron or a set of them work, we are far away of understanding the brain in its fullness.

Instead, **Weak Artificial Intelligence** stands for an AI designed to solve a specific problem. Even AlphaGo [SHM+16] [SSS+17], a computer program developed by DeepMind and one of the most important accomplishments in AI research due to its complexity and innovation, is considered to be a narrow AI (Weak AI).

Moving away from Weak Artificial Intelligence to AGI is a complex journey but neural networks are easing it up. DeepMind has also recently submitted a paper called *PathNet: Evolution Channels Gradient Descent in Super Neural Networks* [FBB+17] which could be the stepping stone of first artificial general intelligence.

Artificial intelligence is not a recent concept. The first work that is now generally recognized as AI was [MP43] formal design for Turing-complete artificial neurons, dated in 1943, and the field of AI research was born at a workshop at Dartmouth College [JMS18] in 1956.

So, what conditions are different that have lead to a recent growth in AI research? Essentially three factors, almost unlimited computing power, more efficient algorithms and enormous amount of data available. The work in this project will be mainly focused on the second concept and the development will be around last innovative AI publications.

1.2 Autonomous UAVs

Research in the field of Unmanned Aerial Vehicles (UAV) is known to be one of the most interesting races in both academic and industrial environments. Considering its multiple applications including package delivery, aerial photography, industrial inspection, surveillance, zone control or monitoring and much more, there is a broad space for innovation and progress [VV14].

UAVs distinguish themselves from classic aircraft in being controlled by a remote operator instead of a on-board pilot. UAVs are usually guided by GPS signals which involve map errors and uncertainty. Instead of the requirements of human remote interaction, artificial intelligence is recently rubbing the point of achieving completely autonomous vehicles by the time this project is being written. The fundamental aspect of autonomous vehicle guidance is

trajectory planning. Historically, two fields have contributed to trajectory or motion planning methods: robotics, and dynamics and control [GKM10].

The main difference between automatic and autonomous control is that, in the first one, the engineer designing the goal needs to perfectly define everything beforehand in order for the UAV to solve it. Instead, in autonomous control, the UAV learns how to achieve the goal itself, with artificial intelligence, without predefining how to achieve the task, and the human interaction is only required in order to give the UAV the sufficient information to learn. **Reinforcement learning (RL)** methods, the core of this project, enable a vehicle to autonomously discover an optimal behaviour through trial-and-error interactions within the environment it is surrounded by.

Nevertheless, memory and computational complexity constraints affect reinforcement learning scalability. Throughout this project, the use of **deep learning** will arise in order to address and overcome this issues, enabling reinforcement learning methods to scale to problems that were previously intractable. The use of deep learning algorithms together with RL defines the field of **deep reinforcement learning (DRL)**, the enabler of this work.

1.3 Baseline of this work

In order to set up the objectives and goals of this project, it must be clearly distinguished what were the previous accomplishments regarding past research. Over his master project [Ker18], Kjell Kersandt built up the first prototype of a DRL framework for autonomous UAVs.

Supervised also by Cristina Barrado, the project ended up in a huge success. Besides building and setting up the framework for UAV simulations, final results came near human performance, in a similar comparison to what DeepMind published in [MKS$^+$15], being it the major source for the development of the project. During this work, simulations were performed in a narrow custom environment, with a fixed altitude for the UAV and only one simple task to perform. While the overall performance remained below the human-level of comparison, potential improvements on several aspects that could lead to even higher reliability and finally a superhuman performance, were clearly identified.

The project covered the state-of-the-art DRL theory and perfectly reported a detailed research comparison on available implemented algorithms, enabling further research to be easier and more comprehensible.

1.4 Project objectives

The main objective of this project, is and has been during its entire development, to learn. Learning how we, as humans, can build intelligent systems able to automate a task and perform it at a better level than our capacities.

At a more practical level, this project clearly involves different objectives. The two major ones are scaling the baseline work to a near-real simulated environment, and enhancing the current architecture in order to solve an autonomous task at a near-human performance.

Research is focused on the joint of computer vision and deep reinforcement learning, understanding and evaluating how to improve the way neural networks perceive and gather more information in DRL methods, rather than the mathematical algorithm itself. Among all this, the project involves also a lot of dedication in software development plus long training and testing simulations, enabling the UAV to behave in a more realistic way.

Other more technical objectives include a research in timing versus efficiency in deep learning models, checkpoints of best training models, geofencing intelligence, real control inside UAVs dynamics and much more which will be better explained in the course of the project itself.

1.5 Project outline

In order to better describe what will be presented in each chapter of this project, an outline is provided emphasizing the important aspects of each one:

Chapter 2 (Reinforcement learning)
: introduces mathematical and theoretical concepts of reinforcement learning. It focuses on the previous knowledge one must acquire in order to comprehend future advances above historical algorithms. It does not intend to cover all methods of reinforcement learning but to overview the ones required for the project purpose.

Chapter 3 (Deep reinforcement learning)
: the chapter comes up with the matter of the project, deep reinforcement learning. It explains the DQN algorithm published by DeepMind, a Google company, displaying also a personal analysis on the results obtained. Additionally, it presents the Double DQN, an improvement on the base algorithm.

Chapter 4 (Materials and methods)
: summarizes how the development of the project has been made, what tools have been used and what research has been focused on. In an organized and brief manner, it shows

how all the methods and tools integrate together in an unique framework. Reshuffles from scratch the way to approach the initial task and provides a totally new architecture based on the improvements and extensions proposed beforehand.

Chapter 5 (Results)

displays the final results with an extended comparison. Ultimately, evaluates the direction and track for future research in the field.

Chapter 6 (Conclusions)

closes the project with an evaluation of the final and most important aspects together with analyzing the accomplishment of the initial objectives.

Chapter 2

Reinforcement learning

This chapter is structured in the basic concepts for understanding reinforcement learning problems, but at some point it directly sticks to the most important pieces for this thesis. Noticeably, reinforcement learning is a complex and broad area in which we do not have enough opportunity to overview all methods and approaches. The best way for immersing into reinforcement learning is taking a look at [SB98] and the previous "state-of-the-art" work of this thesis carried in [Ker18].

2.1 Introduction

Supervised learning is nowadays the most used type of machine learning, a particular subfield of artificial intelligence. In supervised learning, the learner is typically provided with two sets of data, a training set and a test set, yet sometimes part of the training data is used as a validation set. To actually learn, the idea is to provide the learner with a set of well labeled examples in the training set. As labeled is meant the fact that they have been previously well classified by humans. The goal for the learner is to develop a rule to classify unlabeled examples in the test set with the highest possible accuracy and the lowest possible loss.

From speech automated systems, mail spam detection, weather prediction or hand-writing recognition to image classification are just some of the clear examples of supervised learning. However, for many problems of interest, the paradigm of supervised learning does not provide enough flexibility to solve the problem.

Whilst supervised learning problems receive an instructive feedback, in other words, it tells you how to achieve your goal, in reinforcement learning problems, an evaluative feedback is received, telling the agent how well it has achieved the goal.

In the image classification example, an instructive feedback is received. When the algorithm developed attempts to classify a certain image sample, it is directly told what true

class is. If an evaluative feedback had been returned, the classifier would have received a certain score in classifying the image, e.g. $+20$ points. But, without any more context, what does receiving $+20$ points means?

This evaluative process leads to the ability of implementing a more complex, intuitive and accessible system. Maybe those $+20$ meant that a good decision was made, or further iterating with the environment we find that it was a poor score.

2.2 Agent-environment scheme

Reinforcement learning problems are based in learning how to achieve a certain goal from interaction. The most important components are the *agent*, which is basically the learner and decision-maker, and the conditions or surroundings with whom the *agent* interacts, formerly called, the *environment*. This interaction is carried during a sequence of discrete time steps t.

At each time step, the *agent* is in state s_t from the state space $\mathcal{S}$ and chooses which action a_t to take from the set of possible actions in the action space $\mathcal{A}(s_t)$. The *environment* then responds with a new state s_{t+1} and a numerical reward r_{t+1}.

As it will be seen later, the *agent* can choose which action to take in a given state, which has a significant effect on the next state it will see. However, the agent does not control the dynamics of the environment completely.

By taking a look at Figure 2.1, the general idea can be understood. The states are the basis for the *agent* to make the choices, the actions are the choices itself and the rewards are the basis for evaluating these choices.

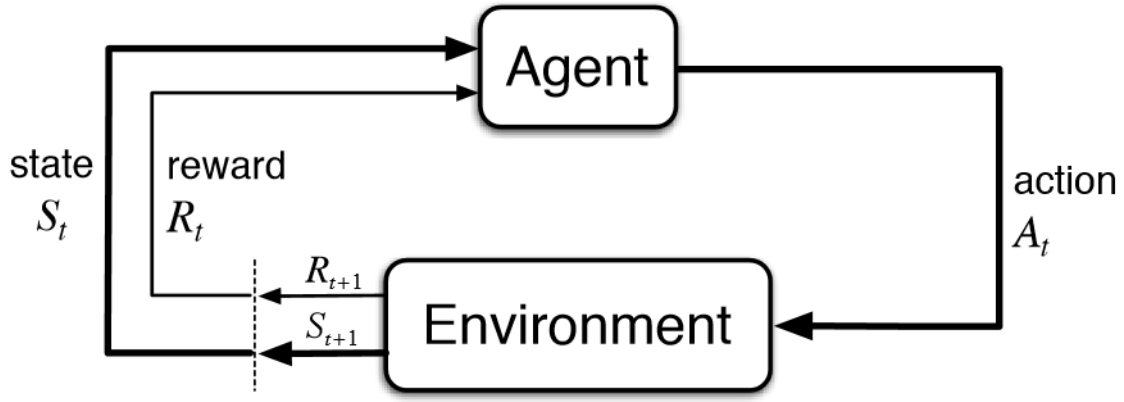


Figure 2.1: Reinforcement learning basic scheme, the agent-environment interaction.

In the end, the ultimate goal of the *agent* is to, at each time step t, map from states to probabilities for selecting each possible action. This mapping is called the *policy* and is denoted π_t, where $(\pi_t(a|s)$ is the probability that $a_t = a$ if $s_t = s$.

$$\pi_t(a|s) = P(a_t = a|s_t = s) \tag{2.1}$$

2.3 Markov Decision Process

RL states satisfy the Markov property:

A state s is said to satisfy the *Markov property* if the future is independent of the past given the present. In other words, the current state s describes all the past states and is sufficient for successive states. In probability theory and related fields, a *Markov process*, is a stochastic process that satisfies the *Markov property* [Ser09] [ER12].

A *Markov process* (or *Markov chain*) is a tuple $\langle S, P \rangle$

- S is a (finite) set of states
- P is a state transition probability matrix,

$$P(s'|s) = Pr[S_{t+1} = s'|S_t = s] \tag{2.2}$$

More in depth, a *Markov reward process* is a *Markov process* with value judgments. This basically shows how much reward is accumulated across some particular sequence that is sampled from a *Markov process*.

A *Markov reward process* is a tuple $\langle S, P, R, \gamma \rangle$

- R is a reward function,

$$R_s = \mathbb{E}[R_{t+1}|S_t = s] \tag{2.3}$$

- γ is a discount factor,

$$\gamma \in [0, 1] \tag{2.4}$$

Equation 2.3 shows that when being in a state s, how much reward do we get for being in that state only. It is just the immediate reward. But actually, what we look forward to in

reinforcement learning is the cumulative sum of these rewards, noted as G_t in equation 2.5, referencing the *goal*.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.5}$$

The discount factor γ sets up the balance between immediate and future rewards. A factor near 0 will make the agent "myopic" by only considering immediate rewards, while a factor approaching 1 will make it strive for a long-term high reward, more "far-sighted".

With all this information, we are able to randomly sample those transitions and compute, but in this scheme there is not any agent making decisions. In order to solve the reinforcement learning problem, we need to look for one more piece of complexity, actions. And all this sets up what is called a *Markov decision process* (MDP).

A *Markov decision process* is a tuple $\langle S, A, P, R, \gamma \rangle$

- A is a finite set of actions,
- P is a state transition probability matrix,

$$P(s'|s,a) = Pr[S_{t+1} = s'|S_t = s, A_t = a] \tag{2.6}$$

- R is a new reward function,

$$R_s^a = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \tag{2.7}$$

Clearly, both the transition probability matrix and the rewards now depend on which action is taken. So, where you end up, depends on the chosen actions throughout the process. Formally, there will be one separate P for each action.

2.4 Value functions

The value function is used for the agent to estimate *how good* is to be in a particular state. It is defined with respect to a policy, since the rewards the agent can expect to receive in the future depend on the action it takes. The value of a state s under a policy π is the expected reward

when starting in s and following π and it is denoted as $v_\pi(s)$, called *state-value function for policy* π and defined as

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \tag{2.8}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable. In a similar way, q_π is defined as the *action-value function for policy* π for taking action a in state s under policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]. \tag{2.9}$$

The *Bellman equation* is the formal expression of the recursive relationship between the value of a state and the value of its successor states that value functions satisfy induced by the Markov property. The *Bellman equation* for v_π is formally expressed in 2.10 equation,

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')], \forall s \in \mathcal{S}. \tag{2.10}$$

The equation states that the value of the starting state must be equal to the discounted value of its expected next state plus the reward, averaging over all the possibilities and weighting each of them by is occurrence probabilities.

Noticeably, the objective of RL is to find the policy that maximizes the reward over the long run. In evaluating policies, policy π is considered to be better or equal to a policy π' if the expected return of that policy is greater than or equal to that of π' for all states. For finite MDPs, there is at least one policy that fulfills this previous statement against all other policies, called *optimal policy* and denoted π_*.

In consequence, the *optimal state-value function* is denoted v_* and is defined as:

$$v_*(s) \doteq \max_\pi v_\pi(s), \forall s \in \mathcal{S} \tag{2.11}$$

The same goes for the *optimal action-value function*, denoted q_* and defined as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{2.12}$$

Taking this into account, equation 2.10 can be written into the *Bellman optimality equation* for both v_* and q_* [SB98].

2.5 Classifications

Environment knowledge

Finite MDPs are distinguished between *model-based* and *model-free* problems. A *model-based* approach has full knowledge of all possible states $\mathcal{S}$, actions $\mathcal{A}(s_t)$, state-transition probabilities $P(s'|s,a)$ and immediate rewards r_{t+1}. Noticeably, a *model-based* problem can be solved by algorithmic planning prior to simulation.

Clearly, this is highly limited and only leads to fully-knowledgeable processes. However, actual reinforcement learning methods are able to solve problems without previous knowledge about the environment. Those are called *model-free* problems, where the agent has to gather experience by interacting with the environment. This approach is excellent for enabling UAVs to perform autonomous tasks and adapt to new environments.

Exploration vs exploitation

Since reinforcement learning was developed so as to emulate human learning styles, this explanation gets easier by figuring an example. In life, we learn a lot through the years. Since we are young, education teaches us to search for our career options, to both find what we like and see what we are good at.

Two ways to face this search can be adopted. You can *explore* more and always look for different options, or you can settle on an option, *exploiting* what you already know. The same goes for RL. When the agent explores more, it takes risks in the process and exposes to more failures. But when it stops exploring, it settles on something, and it is possible that this is not the best option and there is something better out there. The agent risks not searching for other option that could be potentially more beneficial in the long run.

This trade-off has existed during decades and will still exist, maybe because there is not a certain solution and it always depends on the situation. Later in this chapter, an approach to solve this dilemma for our system is introduced.

On-policy vs off-policy

In order to further address the problem introduced above, two different approaches are grouped, *on-policy* and *off-policy* learning methods.

In *on-policy* learning, the agent commits to always have an exploring part, and therefore tries to find the best policy that still explores. On the other way, *off-policy* methods use two different policies, one that is learned about and becomes the optimal policy, called *target*

policy π, and one that is more exploratory and is used to choose the actions and generate behaviour, called *behaviour policy* μ.

Both learning approaches hold their advantages and drawbacks, and it also depends on the explicit algorithm used. This project will be focused mainly in a type of *off-policy* learning method due to the recent research progress over these areas.

2.6 Temporal difference learning

Temporal difference (TD) methods are *model-free* like *Monte-Carlo* methods and update estimates on the basis of learned estimates, analogous to *dynamic programming*. Both concepts are extensively explained in [SB98]. Unlike *Monte-Carlo* methods, TD methods do not have to wait until the end of an episode to determine the increment to $V(S_t)$ with the actual return G_t.

Instead, TD methods just wait until the next time step and use an *estimated* return equal to $R_{t+1} + \gamma V(S_{t+1})$. With the immediate reward R_{t+1} and the estimate for $V(S_{t+1})$, TD methods directly form a target and make an useful update. The value function $V(S_t)$ update toward the estimated return $R_{t+1} + \gamma V(S_{t+1})$ is defined as:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.13}$$

where α is the *learning rate*. The difference between the estimated value of $V(S_t)$ and the following better estimate $R_{t+1} + \gamma V(S_{t+1})$, which is based on the agent's immediate experience, is called *TD error* and denoted δ_t:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2.14}$$

So, as *Monte-Carlo* methods, TD methods follow the pattern of generalized policy iteration [VDW78] but in *Monte-Carlo* the error obtained is equivalent to the sum of all TD errors for every time step until the end of the episode. Depending on the application and number of steps per episode, delaying all learning until the end of a long episode introduces a high variance, therefore can be critical for some systems.

TD methods open up a new paradigm because they can be implemented in an online, fully incremental way and several resources have shown a much faster convergence than MC ones [Sut88].

2.7 Q-learning

In order to explain the *Q-learning* algorithm, which is the theoretical base of this thesis, a few premises are required to be introduced beforehand.

Action-value function

When facing *model-based* problems, state values are sufficient to find an optimal policy. However, in *model-free* problems, having the optimal action-value (values of state-action pairs) function q_* avoids the agent to do a one-step-ahead search. This is because q_* effectively catches the results of all one-step-ahead searches: for any given state s it is able to find an action that maximizes $q_*(s,a)$. In other words, if you are using *model-free* learning, it is better to estimate action values $Q(s,a)$ instead of the state values $V(s)$.

The action-value function allows optimal actions to be selected without knowing anything about possible successor state values and ables TD learning to converge faster to an optimal policy [SB98].

ε-Greedy: Behaviour policy

In order to maintain the exploration versus exploitation trade-off, the use of ε-*greedy policy* [TP11] as a behaviour policy was reported to be hard to beat [VM] and often the method of first choice [SB98] instead of more complex methods.

It relies on a simple but effective method, in which the amount of exploration is globally controlled by a parameter ε. Its main advantage is the suitability for large or even continuous state-spaces, due to no memorization of exploration specific data is required. Equation 2.15 formalizes the ε-greedy policy, $\mu(S_t)$.

$$\mu(S_t) = \begin{cases} \text{argmax}_a\, Q(S_t, A) & \text{at probability } 1-\varepsilon \\ \text{random } A & \text{at probability } \varepsilon \end{cases} \tag{2.15}$$

The algorithm

Q-learning algorithm introduces a fully implementable TD method within an off-policy approach. At each state s_t the actual action A_t is chosen with respect to the behaviour policy μ. But alternatively, Q-learning considers an alternative successor action A', which would

have been selected with respect to the target policy π. The action value for starting state s_t with action A_t is updated towards this alternative action A':

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1},A') - Q(S_t,A_t)] \tag{2.16}$$

Since the target policy π improves greedy with respect to $Q(s,a)$ and the behaviour policy μ improves with exploratory ε-greedy with respect to $Q(s,a)$, the TD control Q-learning algorithm [WD92] is formulated as:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)] \tag{2.17}$$

As deduced from (2.13), the quantity in brackets expresses the TD error δ_t of the Q-function:

$$\delta_t = R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t) \tag{2.18}$$

The main idea is that the Q-learning method evaluates if the action taken by the behaviour policy is better or worse than the target policy, which approximates the optimal action-value function $q*$. Q has been shown to converge to $q*$ with probability 1 [Sut88].

The original Q-learning pseudo-code (in Algorithm 1) summarizes all the concepts discussed above:

Algorithm 1 Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a)$ randomly
Initialize discount factor γ
Initialize step-size parameter α
repeat (for each episode):
    Observe initial state S_1
    repeat (for each step of episode):
        Choose action A_t using policy derived from Q (e.g. ε-greedy)
        Take action A_t
        Observe reward R_{t+1} and new state S_{t+1}
        $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)]$
        $S_t \leftarrow S_{t+1}$
    until S_t is terminal
until end of learning

Q-learning has been classified as an adaptive control algorithm that is able to converge online to the optimal solution for completely unknown systems [LVV12].

Chapter 3

Deep reinforcement learning

3.1 Neural networks in RL

Using deep learning with reinforcement learning is what sets up deep reinforcement learning. The use of neural networks in reinforcement learning as function approximators for high-dimensional inputs was introduced by Paul J. Werbos, the scientist who first described the process of training neural networks trough backpropagation errors. In [Wer89] Werbos trains a neural network by error backpropagation to learn policies and value functions using temporal difference algorithms.

Although for Werbos scenario worked well, neural networks have shown to provide instability or divergence to reinforcement learning problems. A deeper explanation can be found at the deadly triad issue in [SB98] and a more mathematically conclusion in [TVR97].

Recent publications induced huge progress and have led to a broad path in direct applications and further research, showing neural networks can be also extremely powerful in this area of machine learning.

In reinforcement learning, the agent uses neural networks to learn to map state-action pairs to rewards. Neural networks use coefficients to approximate the mapping function relating inputs to outputs. The learning phase aims to find the right coefficients, also known as *weights* by iteratively adjusting them along gradients that obtain less error.

Convolutional neural networks (CNN) are typically used in supervised learning, where the network applies a label to an image, basically by ranking the labels that best fit the image in terms of their probabilities. A good example is shown in Figure 3.1, identifying whether an aerial image obtained by a drone is likely to be of a private property or not [GM18], just for the infinite number of possibilities it holds.

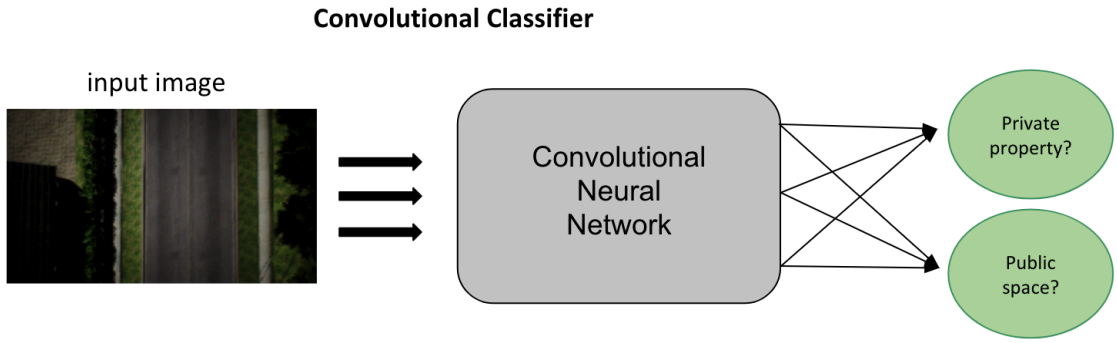


Figure 3.1: Example of a binary image classification problem using a convolutional neural network in order to decide whether a drone is flying over a private propierty or the public space.

In the case of reinforcement learning, the convolutional neural network used in supervised learning to label images, is used to rank the possible actions of a given state. In other words, it helps to approximate the agent *policy* π_t, mapping states to the best action, taking into account $Q(s,a)$, that as we stated before, maps state-action pairs to the highest combination of immediate rewards with all future rewards that might be gathered by later actions in the trajectory. With the expected rewards assigned, the Q function will basically select (when acting greedy) the state-action pair with the highest Q value. A basic approach is what Figure 3.2 represents.

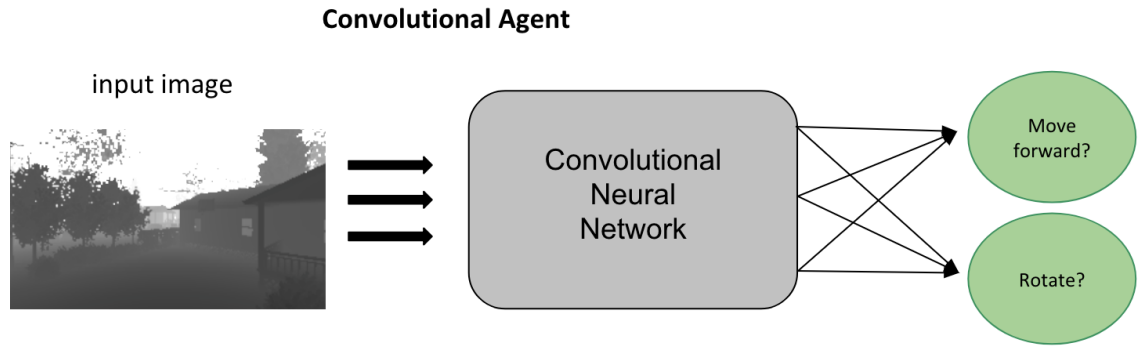


Figure 3.2: Example of an agent that following an optimal policy maps each state (represented by an image) into two possible actions, moving forward or rotating.

The main difference relies on how the neural network adjusts its weights. In supervised learning, the problem begins with the knowledge of the labels that the neural network will try to predict. The mapping is directly done from images to labels taking into account the backpropagation error of the network. For the case of reinforcement learning, neural network coefficients will be initialized stochastically, and using feedback from the environment the neural net will use the difference between its expected reward and the obtained reward to adjust its weights and improve the state-action knowledge.

Noticeably, reinforcement learning problems set up more restrictions than a simple supervised learning one, basically because it relies on the environment to send a scalar number in response to each action carried by the agent. This seems obvious, as it is how reinforcement learning works. But the rewards are delayed and affected by unknown variables introducing noise to the feedback loop. The neural network will need to adapt to a more complex expression of the Q function. It will take into account not only the immediate rewards produced by an action, but also the delayed rewards returned several time steeps later in the sequence.

3.2 Deep Q-Network

Risen from the inspiration of DeepMind recent research publications, we used Keras [C+15], as it will be later explained along Chapter 4. in order to build a similar neural network architecture for our project. As stated by DeepMind, the overall goal is to use a deep convolutional neural network to approximate the optimal action-value function

$$Q_\pi(s,a) = \max_\pi \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi] \tag{3.1}$$

which represents the maximum of the sum of rewards r_t discounted by γ at each time-step t, achievable by a behaviour policy $\mu = P(a|s)$, after making an observation (s) and taking an action (a) as previously explained in the Q-learning section.

The release of the DQN (Deep Q-Network) paper by DeepMind [MKS+15] noticeably changed Q-learning introducing a novel variant with two key ideas.

The first idea was using an iterative update that adjusted the action-values (Q) towards target values ($\gamma \max_a Q(s_{t+1}, a)$) that were only periodically updated, thereby reducing correlations with the target.

The second one was using a biologically inspired mechanism named experience replay that randomizes the data removing correlations in the observations of states and enhancing data distribution, with a higher-level demonstration and explanation by previous research in [MMO95], [OPBDC10] and [Lin93]. The use of the experience replay encourages the

choice of an off-policy type of learning, such as Q-learning, because if not, past experiences would have been obtained following a different policy from the current one. Two huge advances can be taken out from this, one is that each training batch consists of samples of experience obtained randomly from the stored samples and current experience, so temporal correlation is clearly avoided. The other one is that each step in the agent's experience can be used in many weight updates, so a significant gain in efficency is obtained in learning from the environment.

The whole process consists in parameterizing an approximate value function $Q(s,a;\theta_i)$ using the CNN shown in 3.3, in which θ_i are the weights of the Q-network at iteration i. For the experience replay, agent's experiences $e_t(t_t,a_t,r_{t+1},s_{t+1})$ are stored at each time-step t in the replay memory $\mathcal{D}\{e1,\ldots,e_N\}$, where N sets the limit of entries, with the possibility of replacing older experiences for new ones when the limit of the memory is reached.

The standard Q-learning update for network parameters θ after taking action A_t in state S_t and observing the immediate reward R_{t+1} and resulting state S_{t+1} is

$$\theta = \theta_t + \alpha[y_t^Q - Q(S_t,A_t;\theta_t)]\nabla_{\theta_t} Q(S_t,A_t;\theta_t), \tag{3.2}$$

where y_t^Q is the estimated return and is defined as Q-target:

$$y_t^Q = R_{t+1} + \gamma \max_a Q(S_{t+1},a;\theta) \tag{3.3}$$

This update resembles stochastic gradient descent, updating the current value $Q(S_t,A_t;\theta_t)$ over the TD-error towards a target value y_t^Q.

As exposed in [MKS+13] and [MKS+15] the agent was evaluated into the Atari 2600 platform. It offered a diverse array of tasks designed to be difficult and engaging for human players. Using a reinforcement learning feedback and stochastic gradient descent in a stable manner together with large neural networks incredibly outperformed the best existing reinforcement learning methods on 43 out of the 49 games without incorporating any additional prior knowledge.

The trained agents were compared to professional human testers scores playing under controlled conditions. The publication showed that the DQN agent achieved more than 75% of the human score on more than half of the games (29 out of 49).

To comprehend and summarize all the concepts, a practical overview is useful. Taking into account DRL is living through a massive research hot topic and every week new publications appear to change and enhance previous methods, displaying clearly the main ideas behind [MKS+13] and [MKS+15] will help new readers.

All in all, DQN uses experience replay and periodically updated Q-targets given:

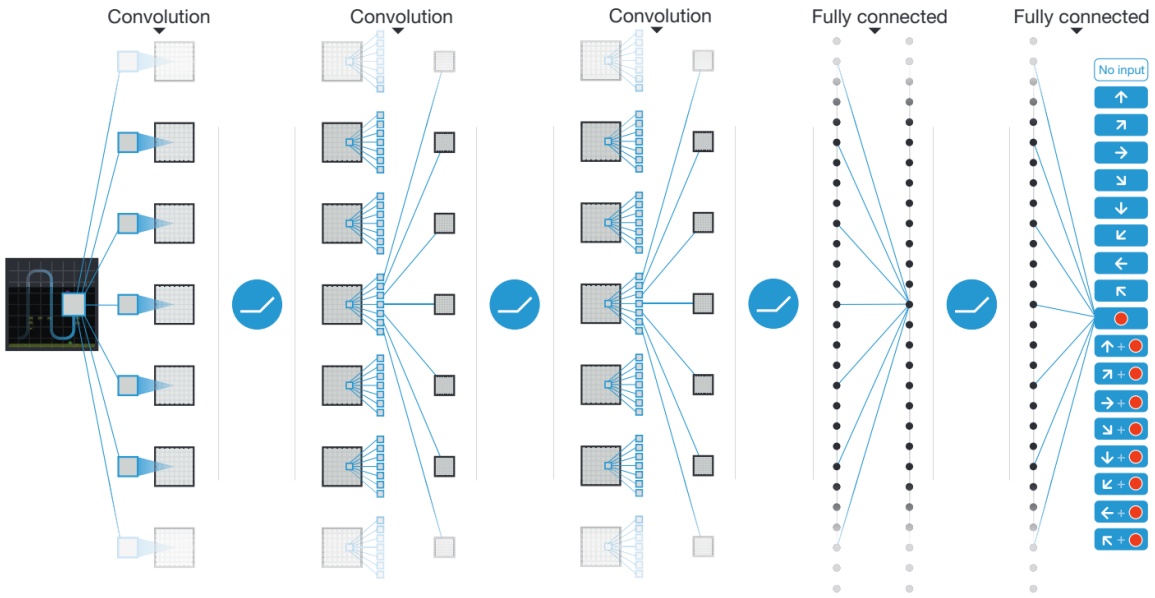


Figure 3.3: CNN architecture for DeepMind DQN playing Atari games [MKS+15]. The input to the neural network consists of an 84 X 84 X 4, followed by three convolutional layers and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (ReLU)

- End-to-end learning values of $Q(s,a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s,a)$ for 18 joystick/button positions
- Reward is change in score for that step
- Take action a_t according to ε-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
- Sample random mini-batch of transitions (s,a,r,s') from $\mathcal{D}$
- Using variant of stochastic gradient descent
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

$$\mathscr{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}[(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s,a;w_i))^2] \tag{3.4}$$

With a little more detail, our Figure 3.4 shows the published performance in games among three main blocks: Games in which DQN achieves a better performance than the human tester score, the ones in which a similar performance (between 80-120%) is obtained, and the rest, which is the biggest group, in which the human tester achieves a better score in the games than the DQN algorithm. After all, the assumption extracted from the publication that "the DQN agent achieved more than 75% of the human score on more than half of the games (29 out of 49)" is true, but when paying attention to the real values segmentation, human testers clearly obtain a better overall performance.

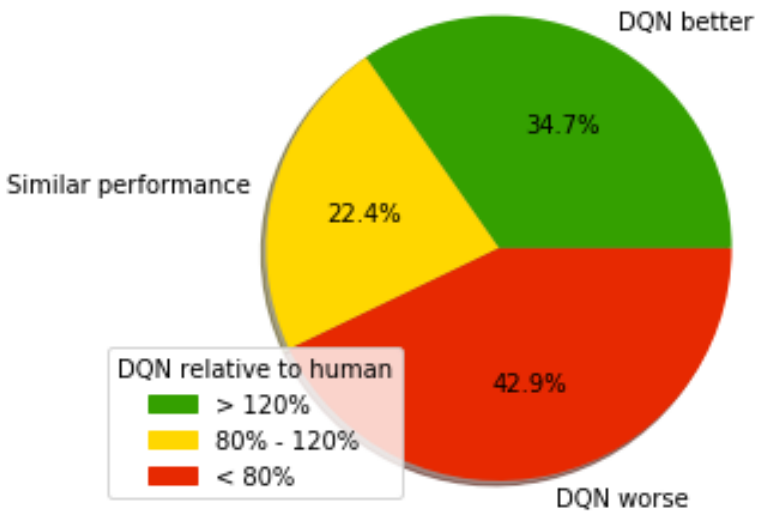
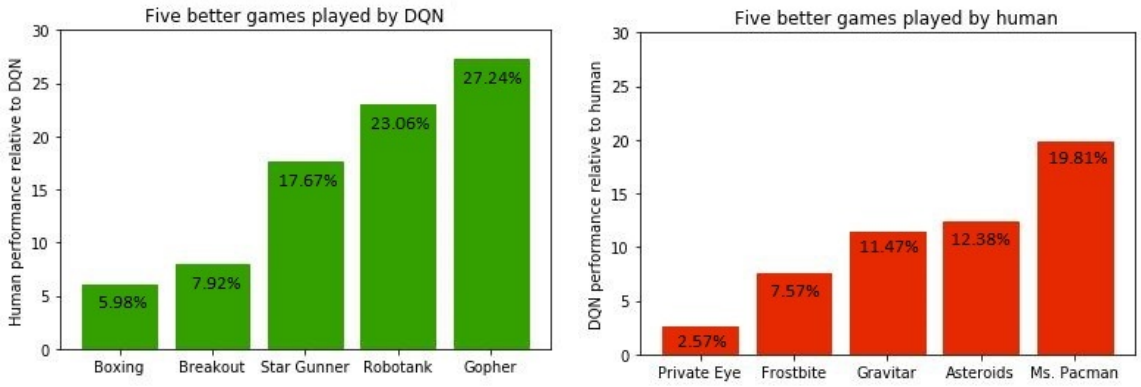


Figure 3.4: Pie chart exposing a comparative analysis on game performance for DQN and human testers.

The following graphs in Figure 3.5 exhibits the performance results of 10 games out of the total 49. These games are chosen as the best and the worst five for DQN relative to the human scores.



(a) The percentage shows the human performance compared to the DQN best five played games

(b) The percentage shows the DQN performance compared to the human best five played games

Figure 3.5: Colors are chosen relative to green seen as a better performance for DQN and red seen as a worse performance for DQN.

3.3 Double DQN

Using the Q-learning algorithm results in a positive bias by definition due to the maximum of the estimates is used as an estimate of the maximum of the true values, making it likely to select overestimated values using a greedy policy as the target policy. From 3.3 one may realize it uses the same samples both to select the maximizing action and to estimate its value, which results in a slower convergence to the optimal policy [SB98]. The idea proposed in [Has10] and named as *Double Q-learning* is basically based in decoupling action selection from evaluation.

Two action-value functions Q_1 and Q_2 are learned by assigning each experience randomly to update one of the two function with the two sets of weights, θ and θ' in Double Q-Learning, one set of weights is used to determine the greedy policy and the other its value. First, the authors rewrite equation 3.3 as

$$y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \theta_t); \theta_t). \tag{3.5}$$

And the two Double Q-learning targets can then be written as

$$y_t^{DoubleQ_1} = R_{t+1} + \gamma Q_2(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q_1(S_{t+1}, a; \theta_t); \theta'_t) \quad \text{and} \tag{3.6}$$

$$y_t^{DoubleQ_2} = R_{t+1} + \gamma Q_1(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q_2(S_{t+1}, a; \theta'_t); \theta_t).. \tag{3.7}$$

As referenced, Q_1 is used to determine the maximizing action $A_* = \text{argmax}_a Q_1(a)$ and Q_2 is used to provide the estimate of its value with $Q_2(A_*) = Q_2(\text{argmax}_a Q_1(a))$ shown in (3.6). The second set of weights can be updated symmetrically by switching the roles of θ and θ' into (3.7), achieving unbiased estimates.

As only one estimate is updated per step in a random selection, but two estimates are learned, it doubles the memory requirements but not the computational effort made at each step. The Double Q-learning was extended for the DQN-algorithm in [VHGS16]. Furthermore, the DQN-algorithm provides with the target network θ^- a natural candidate for the second value function, without having to introduce additional networks. The *Double DQN* algorithm remains the same as the original DQN-algorithm, except replacing the target y_{DQN} explained in [Ker18] due to the limited space with

$$y_t^{DoubleDQN_1} = R_{t+1} + \gamma Q_2(S_{t+1}, \underset{a}{\text{argmax}}\, Q_1(S_{t+1}, a; \theta_t); \theta_t^-) \quad \text{resp.} \tag{3.8}$$

$$y_t^{DoubleDQN_2} = R_{t+1} + \gamma Q_1(S_{t+1}, \underset{a}{\text{argmax}}\, Q_2(S_{t+1}, a; \theta_t^-); \theta_t), \tag{3.9}$$

where the weights of the second network θ' of double Q-learning in (3.6) and (3.7) are replaced with the weights of the target network θ^-, performing the update to target network as in neural fitted Q-iteration introduced before. This project revolves around the two mentioned algorithms, DQN, and Double DQN (DDQN).

Chapter 4

Materials and methods

The first approach for our drone framework was carried in [Ker18] and showed incredibly great results. The best performing Double DQN-agent obtained an 80% success rate during evaluation. While the overall performance remained below the human-level of comparison, potential improvements were identified on several aspects that could lead to even higher reliability and a superhuman performance.

Like in any other reinforcement learning proposal, one must set some boundaries and define a given problem and a certain solution. Our approach is to train an autonomous UAV to reach a goal in the minimum amount of time without colliding with any obstacle. This can be extended to reaching multiple goals in a single flight or even more complex things that will be introduced later. For a broad knowledge of how everything works, each improvement to the baseline is summarized and exposed in a simplified way.

The development of this project has been possible due to recent progress in open-source libraries and public repository collaborations. This chapter presents which resources have been used. Obviously, nowadays there is a vastly amount of competence between frameworks and libraries, ergo there are different ways of approaching the same problem and are continuously being updated. For every project requirements and resources there are more efficient solutions than others, and this always has to be analyzed first. Building a unique framework for DRL methods for UAVs, one of the initial motivations of this project is a difficult task to release and maintain. In this chapter we will analyze the tools used in order to build up our framework and its integration, together with the implementation and improvement from the previous baseline. The code is publicly available in Github and open to external contributions. Excluding code for graphs, front-end development and miscellaneous, everything can be found in the same remote repository: https://github.com/guillem74/DRLDBackEnd

4.1 Tools

All the development in this project has been built with Python, motivated by the great data science and machine learning community and both framework's availability and excellence. Python is also a great and easy language for beginners, being simple to code and holding a fast learning curve for software developers when jumping from another language.

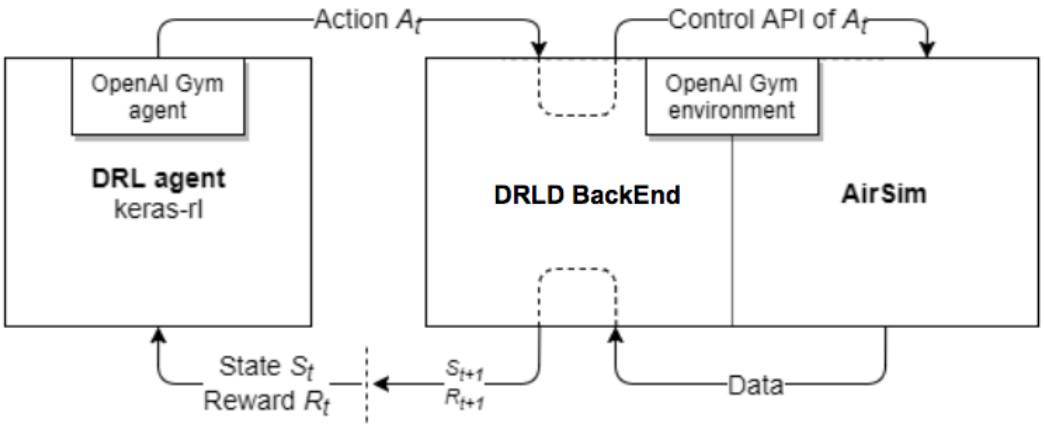


Figure 4.1: Overview of agent-environment interaction using our DRLD BackEnd

In order to visualize an scheme of the whole system, we provide Figure 4.1 which stands the same as in the first version of the framework. Every component will be explained below but one can rapidly figure out how the interaction is built up. As an overview, *AirSim* provides us with a solid UAVs simulator, *OpenAI Gym* brings us a toolkit for building up reinforcement learning algorithms and *keras-rl* acts as a library that allows us to implement deep reinforcement learning algorithms. The first version of our framework was named *AirGym-v0* but we consider it as deprecated due to recent updates in AirSim APIs. In addition, the idea of a name that includes a reference to OpenAI Gym framework limits us when willing to merge into a version that supports multiple frameworks.

4.1.1 AirSim

AirSim [SDLK17] is an open source simulator of vehicles based on Unreal Engine (UE4) from Microsoft AI and Research. AirSim profits from the really good rendering techniques of UE4 brought by the game industry [KG13] bringing strong graphic features and ultra realistic rendering capabilities, see Figure 4.3 as an example. Without the need of being a professional, you can modify existing environments or build your own by implementing objects, blueprints and behaviours in C++, apart from having the access to existing environments.

There are two main motivations for choosing AirSim. The first is their quadcopter realistic features, allowing simulations of existing UAVs such as Parrot AR Drone 2.0 [Par12] inside the game engine with at the same time supporting hardware-in-loop with popular flight controllers such as PX4 for physically and visually realistic simulations. And the other one is that it is basically built up for programmatic control, it exposes APIs so the user is able to interact with vehicle in the simulation programmatically. You can use these APIs to retrieve images, get state, control the vehicle and so on. The APIs are exposed through RPC and accessible via variety of languages including C++, Python, C-sharp and Java. The most important API features for our project are performing UAV complex movements and posing cameras to collect images such as depth, disparity, surface normals or object segmentation. These features make AirSim the strongest option within drone simulators. AirSim is a platform under development and with a large community which really helps it to grow, inserting pull requests into AirSim's public repository or opening issues when errors or improvements come to light.

4.1.2 OpenAI Gym

OpenAI Gym [BCP+16] is a standard framework for setting up reinforcement learning tasks. Gym is a Python toolkit for developing and comparing reinforcement learning algorithms, making no assumptions about the agent's structure in any problem.

Gym aims to help RL research by two main factors:

- `The need for better benchmarks`: While in supervised learning, progress has been driven by large labeled datasets like Imagenet [DDS+09], in RL there is the need to collect large and diverse environments. However, the existing open-source collections of RL environments don't have enough variety, and they are often difficult to even set up and use.

- `Lack of standardization of environments used in publications`: Differences in problem definition, such as the reward function or the set of actions make RL problems complicated to code. This issue makes it difficult to reproduce published research and compare results from different papers.

In other words, Gym lets you create your own environment or use one available. By environment, we do not mean the simulator or game, it means the programmable interaction with it. In Gym you are able to define and obtain, at each time-step, the information needed to piece together and later feed the RL algorithm. The developer has to set up an `Env` class which defines how each step is accounted and enables the reset function. The step function

returns the substantial information for the RL algorithm, meaning the state, the reward, the action taken, if the task is in its terminal state and some extra information. A more detailed explanation can be found at https://gym.openai.com/docs/.

4.1.3 Keras-rl

Keras-rl [Pla16] implements state-of-the-art deep reinforcement learning algorithms in Python and seamlessly integrates with the deep learning library Keras [C+15], working also with OpenAi Gym, making it all more simple. Keras has been gaining space in the Deep Learning industry, being a high-level neural networks API for easy and fast prototyping and the ability to run seamlessly on CPU or GPU (although little things can be done with CPU nowadays). Researchers in both convolutional and recurrent networks prefer to use Keras instead of other low level library, thanks to its modularity and extensibility. For this project, Keras core will be using Tensorflow as the back-end library, but Keras-rl has the possibility to use also Theano or even CNTK.

Keras-rl is built according to the developer needs, giving the ability to define own callbacks and metrics. More in depth, it also provides an easy access to implementing or redefining own algorithms by simply extending some abstract classes. Documentation can be found at http://keras-rl.readthedocs.io/en/latest/. As we stated in the last section, this project revolves around two algorithm extensions, DQN and Double DQN (DDQN), which are going to be the used and implemented branches in Keras-rl. If you take this recommendation, try out and implement also Deep Determinisic Policy Gradient (DDPG) [LHP+15] or Deep SARSA [MBM+16], two upcoming and strong algorithms in reinforcement learning.

4.2 Checkpoints

The main issue from previous work was found to be that during training phase, the random actions taken by the behavior policy created oscillations in the obtained reward. Additionally, a lot of executions were aborted before finishing training and that produced a huge lost on time. An improvement was required in order to train the algorithm for a long period of time and obtain the best possible results achieving high efficiency. Motivated by the inconsistencies of the AirSim simulator, which had been under development as this thesis was conducted, a contribution was demanded in order to keep training phases safe.

A pull request was submitted to keras-rl [Pla16], and it turned into a successfully approved contribution to the open-source library. The code developed was able to check at the end of

each episode whether the agent achieved a better performance than before, and if desired, write the neural network weights into disk before ending the whole training phase.

Checkpoints clearly solved two main problems, which were introduced at the end of the thesis carried in [Ker18]. On the one hand, and the most obvious, was being able to save the current best or even not best, if configured, weights when training was intentionally or accidentally stopped at a certain time-step. On the other hand, and as it will be shown later on, training curves exposed previously to this improvement, clearly exhibited a tendency of not finishing with the best agent's reward efficiency, which induced a worse behavior in testing mode. We provide more information in our recent accepted publication on Digital Avionics Systems Conference (DASC) *here*.

4.3 Agent-environment design

This section will explain the development in each field of the reinforcement learning task separately. It clearly focuses on the environment and the agent features separately, including the available actions, the reward function and the definition of the state.

4.3.1 The environment

The new environment is part of the AirSim v1.1.8 release for Windows binaries build from the Modular Neighborhood Pack in Unreal Engine. The map creates an entire real-world systematic. It includes an entire residential zone, parks with animated winds on trees and a broad range of streets originating a real neighborhood.

(a) Inside environment and front view of the quadcopter

(b) Three-dimensional view over the area

Figure 4.2: AirSim Neighborhood environment views.

Unlike the blocks environment introduced in the baseline, this new environment generate a new issue, related to dimensions. Although not all the environment is captured from

4.2b, because it involves also more green zones, which are not so important for us, this new map covers an area of approximately 770×770 meters, 17 times bigger than the columns environment. This is an enhancement, approaching the simulation to reality, but also has a drawback, taking into account training times. A solution for that problem will be presented in the following subsection.

Data and variables used for the deep reinforcement learning task are presented in Table 4.1. Three new blocks have been introduced, which are going to be discussed from now on.

Table 4.1: Data fetched from the simulation environment at every time-step.

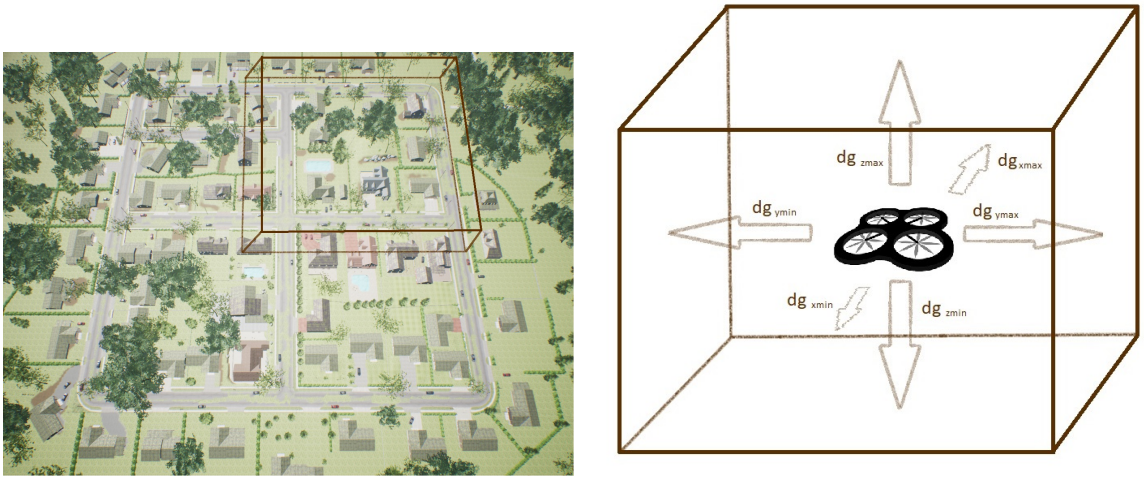
Data	**Meaning**
p_x, p_y, p_z	agent's global x, y and **z position**
d_x, d_y, d_t	**agent's x, y and total distances to goal**
dg_{xmin}, dg_{xmax}, dg_{ymin}, dg_{ymax}	**agent's distances to geo-fence limits**
ψ	yaw angle relative to current orientation
DepthImage	depth image in camera plan (144×256)
arrived	boolean landing info
collided	**boolean collision info**

Geo-fencing

Geo-fencing is known as a method of defining a virtual barrier on a real geographical location. Restricting the flight zone of a UAV has been useful in many cases. Another classic examples are the use of targeting citizens to promote proximity marketing or to send warnings of safety alerts via SMS.

In aerial systems, geo-fencing has been widely used for many approaches. In the area of UAVs, an expert in GPS security [Hum15] and a politician [Sch15b] suggested in 2015 that government regulators should encourage drone manufacturers to build geo-fencing constraints into UAV navigation systems that would override the commands of any operator, preventing the device from flying into protected airspace.

For the case of our simulations, the geo-fencing concept is the most suitable approximation to increase realism. Also, programming a parametrizable geo-fence inside the AirSim simulator [SDLK17] prevents the agent from exploring scenes inside the environment that are not useful for accomplishing a certain task, resulting in a decrease in training time.



(a) Possible geo-fencing for a targeted zone

(b) Top view perspective

Figure 4.3: Geo-fencing limits for our quadcopter.

The idea evolves around the possibility of defining six different fences, two for longitude, two more for latitude and the last ones for altitude at the beginning of each training. Therefore, by adapting both state information and the reward function, the drone has to learn the concept of geo-fencing as another element of the environment; being rewarded negatively when trying to transgress it, and further creating the sufficient knowledge for avoiding it.

Landing

Approximating the simulation towards reality is not a trivial task. Our interaction with the environment needs to be adjusted to real physical dynamics. Drone flights include a takeoff period, the flights itself and a landing period.

Within this approach, the quadcopter takes off to a certain altitude and it remains constant until it reaches the goal coordinates. Nonetheless, it has to fetch altitude position z at every time-step to control, maintain and adjust it when the goal is reached. Upon arriving to the destination, the altitude is slowly diminished until the quadcopter totally lands on the surface, achieving a controlled and stable movement.

Even though a three dimensional dynamism is implemented at this stage, a slightly nearer to three dimension control version is acquired. Recent progress in drones has lead to include the possibility of sending commands from ground-control stations in order to perform discrete actions, such as taking off, landing, and rotating or moving forward at a fixed altitude, which is what we aim to replicate with an autonomous version of it. Obviously, a three dimensional control would imply a fully transferable to real-world format.

Multiple destinations

Expressed at the beginning of the project, one of the main problems of reinforcement learning and robotics in general is environment exploration. Gathering the sufficient knowledge of its surroundings, an agent could be able to take optimal decisions at any given moment.

Nonetheless, general artificial intelligence is not yet achieved and there are many mathematical restrictions on it. On the other side, there are some things which actually enhance the fact of learning more general concepts, in order to later extrapolate from it.

At this project we have already introduced some concepts that boost exploratory issues within deep q-learning. Some of them were using a ε annealed policy or modifying the experience replay size. Other potentially important contributions to general intelligence will be explained at the 5.4.2 subsection, which contains already studied and researched areas but not yet well implemented for this project.

In order to gain more data for the agent, we came up with a more elaborated type of training. While the origin for the UAV was always the same, the destination of the goal changed. The reason of not changing the origin was purely AirSim restrictions. Before the new 1.2 *release*, thrown the 21st of June, the only way of changing the origin of a vehicle was via reading a static JSON at the execution of the binary. So, there was now way to dynamically read it during the same training. Now, thanks to open-source suggestions, Microsoft developers changed it to API calls, making it easier and more accessible.

Therefore, the agent is capable to learn different tasks with different goals within the same training process. This focused more on a general knowledge, rather than the specific accomplishment of a goal, which is basically what a more broad intelligence tries to address. The later analyzed metrics comprise the training to four different destinations, randomly sampled at the beginning of each different episode, which made the task less repetitive and more complex. The chosen destination at each episode is one of the following, $[(137,-48),(59,-15),(-62,-7),(123,77)]$, respectively as always to the *Neighborhood* map. Noticeably, this change affects the geofencing limits, which were adapted in the code. The destinations were chosen to be representative of a real possible destination case.

4.3.2 Actions

In the first baseline, actions were limited to three different ones, going *straight*, and performing a *right yaw* or *left yaw* with different angles to be able to follow more possible directions.

Although we use this set of actions for various testing and results examples, we drastically changed this initial approach for a more realistic one.

Three dimensional actions

This improvement was probably the enhancement that better approaches the project to a real autonomous system, apart from being also the most visual improvement for the behavior of the agent.

Our UAV autonomous agent was originally flying at a fixed altitude, moving only along two dimensions, longitude and latitude and moving forward at constant velocity. Our aim for the UAV is to acquiring three dimensional movements and velocity changes.

The try was giving the agent the ability to increase and decrease velocity within the longitude and latitude coordinates, originating movement in the four cardinal directions (north, east, south, and west), in addition to being able to increase and decrease altitude. However, this displayed great constraints. As the camera of the UAV was always facing the same direction, but moving along other directions, it was not able to 'see' and recognize with what object it was colliding, except when it was a frontal collision. This temporary solution did not provide the system anything rather than drawbacks, moving the agent away from being intelligent. The only way of addressing the issue was placing multiple cameras, one on each side of the quadcopter, which is not an scalable option. Having to place at least four, or six (if you want to see above and below) cameras, and processing their images at real time is clearly a far from affordable computationally expensive idea. Additionally, UAV hardware suppliers are not likely to focus on these features, but the opposite, efficiency in weight and flight aerodynamics.

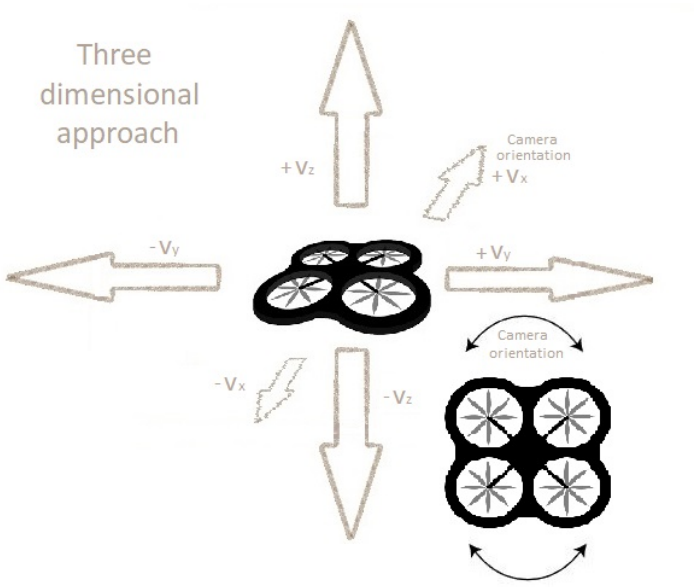


Figure 4.4: UAV possible movements within the environment.

The chosen solution was modifying a function of AirSim Python API in order to increase and decrease velocity values but always facing forward, computing the appropriate yaw angle rotation to perform according to the modified velocity, emulating the behavior of a car or a plane. With these adjustments the agent's actions were reduced to just a general one, which combined, derived in six different actions. The general action is the modification of the velocity on a selected axis x, y or z the equivalent of ± 0.5 m/s. Giving this ability to the agent, it was able to select one of the six different variations at each time-step. Image 4.4 allows to understand better the behavior.

After some tests, the agent was able to move over the environment with total freedom, flying over houses or trees but also going under other certain objects. Moreover, it could reach earlier the desired goals, increasing its velocity, and therefore obtaining a higher reward. Noticeably, this modifications in the action space are correlated with changes in both the state definition and the neural network, which are going to be presented also in this chapter.

4.3.3 Rewards

Motivated by the simplicity and efficiency from the reward function of the first approach, we use a replicated version of it with minimal improvements.

From now on, the geo-fencing limits are taken as a collision, so a reward of -100 is automatically given to the agent when transgressing the fences. Noticeably, the agent needs to match this negative feedback with some information rather than just only capturing depth images, because these virtual fences cannot be seen from its front camera. Later, this will be addressed by always having a knowledge of how close the agent is in respect to the virtual limits.

4.3.4 States

In reinforcement learning problems, the more crucial data is taken into account when taking a decision, the better performance the agent is going to obtain. However, how much data is needed to adequately learn to perform a task? Does it really pay off to acquire knowledge from different inputs or sources? As we will see, it is all a matter of a trade-off. Taking into account computational timing, the difficulty to obtain data, the convergence of the algorithm and the difference on learning, one can evaluate and decide. Three approaches have been developed for delivering the best solution. In this subsection we are going the describe them separately, focusing also in the advantages and drawbacks of each one. Note that in reinforcement learning, the definition of the state and the architecture of the neural network go hand by hand, because one affects the other and vice versa.

Joint state

For this part, the agent's state is not only composed with the 30×100 image part but also with agent's position, distance to the goal and distance to the geo-fence information. Position information is encoded in a 1×2 array with p_x and p_y values. Additionally, information regarding distance to the goal coordinates is encoded in a 1×3 array with d_x and d_y values plus d_t, which represents the Euclidean distance value. Lastly, distance to virtual fences (geo-fencing) are also encoded in the state representation for the agent, composing a 1×4 array with dg_{xmin}, dg_{xmax}, dg_{ymin}, dg_{ymax} values.

All this information is encoded up together setting up the agent's state. So, at each step, all this information is gathered from the interaction between the agent and the environment and being used for the reinforcement learning task.

Velocity state

As in other machine learning sub-fields, overfitting stands formally for *"the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably"*.

The best way to check if our system is overfitted in classical supervised learning is to check if the network is able to classify unseen testing samples into correct the classes. If its performance is poor while has showed great training accuracy/loss values then the model is overfitted. In other words, it is too tightly fitted to the specific data points in the training set, trying to model patterns in the data originating from noise. The same can be applied for DRL, but as testing results are more difficult to analyze, it is more complicated to be solved.

As the expert in the field *Geoffrey Hinton* said "*overfitting is actually a great first step in creating a good neural net*". It means that the neural network is actually learning something. However, when your neural network is overfitted that's when it's time to perform regularization, figuring out if what it's learning can tell you something about unknown data or not.

In order for our system to scale to a real-world application and not only fitted to a certain scenario, the location values $[p_x, p_y]$ of the agent were removed from the state definition, and therefore to the input of the neural network. Learning from the exact location coordinates where you are is not relevant for other environments, and would further lead to misleading comprehensions of it. This also resulted in a reduction of the trainable parameters, leading to a higher efficiency. Moreover, as it will be demonstrated later on, it didn't affect on negatively results and was noticeably an enhancement in learning generalization.

Another decision was to remove the 10×100 image section from the state definition, which significantly reduced by $1/3$ the image dimensions. As this enhancement is more related with the neural network part, it will be explained in Subsection 4.4.2.

Before changing the state definition, the modifications in the action set to moving by velocities enhanced the behavior of the agent, but the obtained results showed it was not good enough. The only reason was that the agent was not able to match its action selection with the change in its state. Thus, we re-defined the agent's state as a 20×100 front depth image together with $[d_x, d_y, d_t]$ distances with goal, $[dg_{xmin}, dg_{xmax}, dg_{ymin}, dg_{ymax}, dg_{zmin}, dg_{zmax}]$ distances with geofencing limits and $[v_x, v_t, v_z]$ velocities in each axis at each time-step. Then, the neural network was able to observe its change in its state in direct form of velocities, when the selected action was already taken. Therefore, it was able to learn from it, as it will be demonstrated later on the results. See also how this affected to the neural network architecture in Subsection 4.4.2.

Time progression in state

When learning from broad environments, the better an agent is able to perceive the changes within it, the higher the chances are its performance will reach optimal levels.

The original standard DQN architecture presented in [MKS+15] actually took as input the last four frames (states) to take a decision. This effectively gave the algorithm a measure of velocity of moving objects. However, using k last states limits the scope of available "memory" and most RL algorithms assume that their environments are Markov, so there is enough information in its current state to predict a future state. Concatenating the frames as suggested, is a relaxation of this assumption. Using recurrent neural networks could be another solution, but would mean re-factoring the whole approach.

However, we decided not to apply it for the image part in due to the high amount of computation required in order to process four frames. Additionally, it would be a huge limitation when transferring the project to the real work, requiring the UAS to process lots of data in near real-time, either doing it by an on-board computation (GPU chips) or at the ground base station.

Instead, we decided to change the state and also provide the neural network with the last four values of velocity, distance to the goal and geofencing distances. This gives the algorithm a sense of the implications of choosing a certain action more progressively. The implementation consisted in building three FIFO (First-In First-Out) queues, one for each input array, holding the last four values (actual value and three past ones). The architecture part is presented in 4.4.3.

Remarkably, the past values were initialized in order not to affect the algorithm. Velocity values were initiated as zero and, destination and geofencing distances started all with origin point values.

4.4 Joint Q-network

Motivated by improving the results of the first approach, a more complex and nearer to real-world application was claimed in order to successfully achieve the demanded tasks. We named this whole approach Joint Q-Network (JQN) derived from the concepts that will be introduced later on.

This new architecture is defined for taking into account the previously defined composed state. This is accomplished by using a *processor* that enables the neural network to handle multiple inputs at every time-step in Keras [C+15]. In this section we are going to introduce the three different approaches of the neural network composition in relation with the previous introduced modifications.

4.4.1 Initial approach

The initial approach of the Joint Q-network is based on the assumptions that the set of actions is the same as in the baseline architecture but the state definition is modified towards the proposal introduced in Subsection 4.3.4. So, at each iteration, data (state information) is separated into two subnetworks (models).

As it is shown in 4.6, the Sequential CNN learns only from the image. The first layer of the CNN part is a 32 kernel 4×4 with stride 4 followed by a 64 kernel 3×3 with stride 2 both using ReLU activation functions. Afterwards, as stated before, a single flatten layer is used to input a tensor of any shape and transform it into a one-dimension tensor but keeping all values in the tensor.

The overall model takes the output of the Sequential CNN model and concatenates it with the information regarding position, goal and geo-fence distances arrays after they are reshaped. The concatenated tensor is then the input of three consecutive 256 kernel dense layers with ReLU activation functions. Finally, the output layer is comprised of a dense linear layer with an output of the action value for each action defined in the system, equal to the output of the model defined in the baseline architecture. You can observe a brief composition of the two connected subnetworks at Figure 4.5.

We named this neural network architecture a JNN model (Joint Neural Network) regarding to using both image and other values to jointly obtain action values. Figure 4.7 presents a

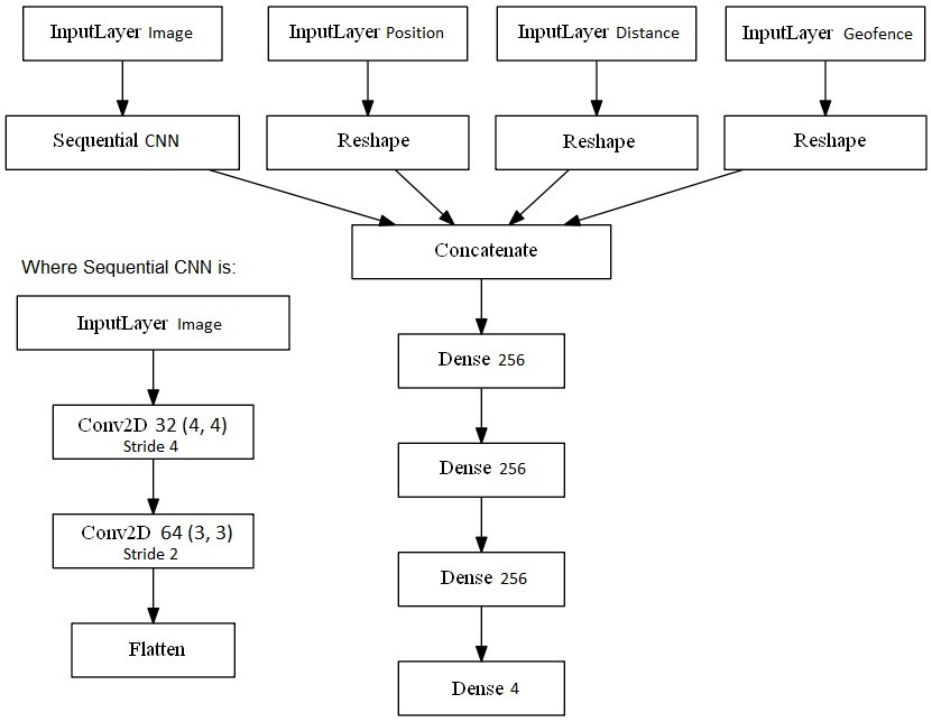


Figure 4.5: Layers of the two models shown in cascade view.

detailed view of all the inputs and outputs of each layer, showing also the learned parameters at each layer.

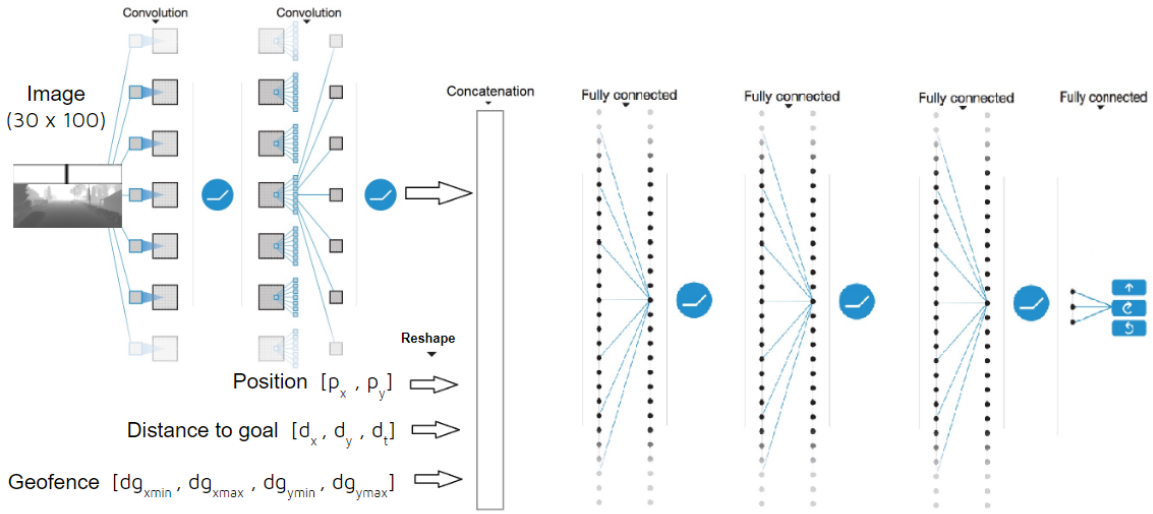


Figure 4.6: Architecture of the multiple input neural network .

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 7, 25)         544
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 15, 3, 64)         14464
_________________________________________________________________
flatten_1 (Flatten)          (None, 2880)              0
=================================================================
Total params: 15,008
Trainable params: 15,008
Non-trainable params: 0
_________________________________________________________________
None
```

```
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
input_1 (InputLayer)             (None, 1, 30, 100)    0
____________________________________________________________________________________________________
input_2 (InputLayer)             (None, 1, 2)          0
____________________________________________________________________________________________________
input_3 (InputLayer)             (None, 1, 3)          0
____________________________________________________________________________________________________
input_4 (InputLayer)             (None, 1, 4)          0
____________________________________________________________________________________________________
sequential_1 (Sequential)        (None, 2880)          15008       input_1[0][0]
____________________________________________________________________________________________________
reshape_1 (Reshape)              (None, 2)             0           input_2[0][0]
____________________________________________________________________________________________________
reshape_2 (Reshape)              (None, 3)             0           input_3[0][0]
____________________________________________________________________________________________________
reshape_3 (Reshape)              (None, 4)             0           input_4[0][0]
____________________________________________________________________________________________________
concatenate_1 (Concatenate)      (None, 2889)          0           sequential_1[1][0]
                                                                   reshape_1[0][0]
                                                                   reshape_2[0][0]
                                                                   reshape_3[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 256)           739840      concatenate_1[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                  (None, 256)           65792       dense_1[0][0]
____________________________________________________________________________________________________
dense_3 (Dense)                  (None, 256)           65792       dense_2[0][0]
____________________________________________________________________________________________________
dense_4 (Dense)                  (None, 3)             771         dense_3[0][0]
====================================================================================================
Total params: 887,203
Trainable params: 887,203
Non-trainable params: 0
```

Figure 4.7: Details on the neural network architecture. The *None* parameter in the Output Shape is used for accepting tensors with dynamic dimensions due to when using *Keras* you can use *Tensorflow* or *Theano* as backend libraries.

4.4.2 Velocity JNN

At this point, we decided to remove the 10×100 image section, which significantly reduced by $1/3$ the image dimensions. This was merely imposed due to two main factors. The first is that the solution of learning a simple value, such as direction towards the goal, from an image, is such a complicated way of doing it. The usage of such amount of data increases the difficulty of the algorithm convergence, and in the case of using CNNs means that convolutional layers will need to successfully detect these features. This resolution meant a relevant reduction in network parameters from 887.203 to 642.214, which clearly enhanced training times and efficiency.

The other main factor was actually that the reason of using such high amount of data to represent the angle towards the goal was a workaround. Before the implementation of the Joint Q-network architecture, only images could be input to the CNN. The usage of this new architecture implies the ability to directly feed the algorithm with scalars, such as the relative and the euclidean distances, as state values for the neural network. The scalar value representing the angle direction was also proposed but results determined it was unnecessary.

Together with the velocity state previously defined and the modified actions of Subsection 4.3.2, the output of the new JNN architecture was $Q(s,a)$ for the 6 available actions. With all previous assumptions, the scheme of the neural network appeared as in 4.8. Network parameters were increased to 642.982, only 768 parameters more to learn.

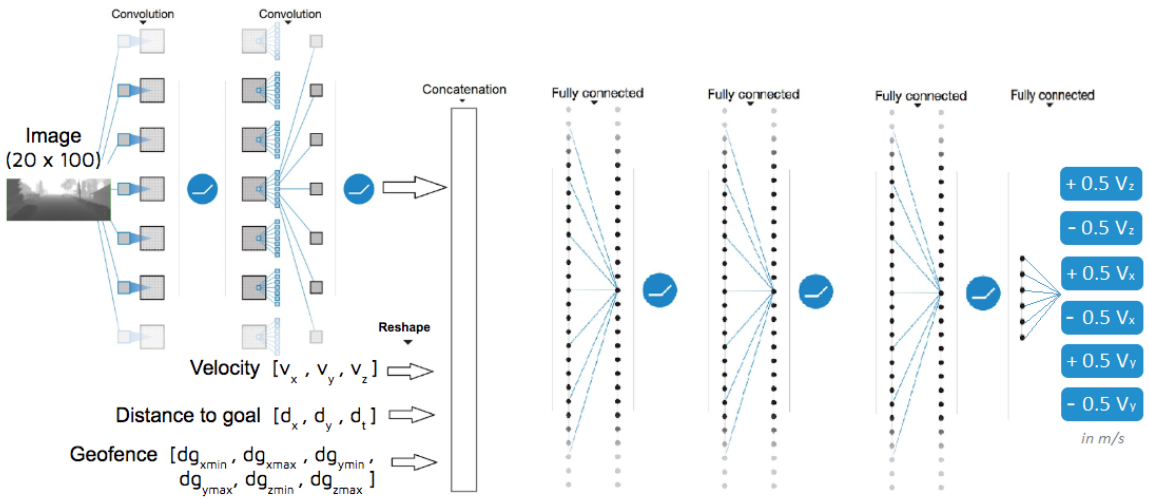


Figure 4.8: Modified JNN architecture for three dimensional UAVs and velocity oscillation.

4.4.3 Time sensitive JNN

Tied to the modifications related to holding a time progression in the state definition, the neural network was again modified, now with the three previous scalar values together with the current one.

Figure 4.9 represents the JNN with the time sensitive approach. Note that each value is expressed in time, which represents the previous explained values (i.e. v_t is equal to $[v_x, v_y, v_z]$, d_t is equal to $[d_x, d_y, d_z]$ and dg_t is equal to $[dg_{xmin}, dg_{xmax}, dg_{ymin}, dg_{ymax}, dg_{zmin}, dg_{zmax}]$ at time t). The same happens with the other values. Taking everything into account, it means a velocity and distance array of 12 values, and a geofencing array of 24 values.

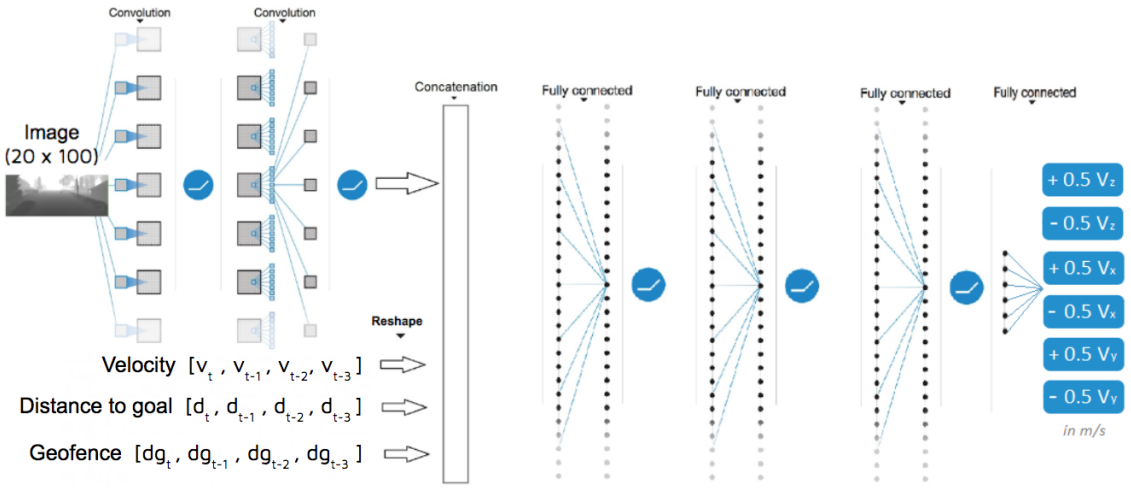


Figure 4.9: JNN architecture with time progressive values.

In summary, parameters increased to 652.198, more than 9.216 additional parameters to learn for the neural network. This means just one percent increment in the number of parameters, but the difference in physical limited hardware is huge.

4.5 Front-End prototype

Enabling all these pieces together has been one of the tasks during the development of this whole project. Using the same coding language and actual framework compatibility facilitates a lot but there has been an underlying work coupling all these tools to work. More than explaining how we have actually integrated everything, we like to focus on what could be done in order to provide less experienced users the tools to try and test the framework. If you want to actually learn the whole interaction you just have to dive into the public repository

and start playing with it, because it cannot be well described in this project document, which is merely focused on research.

The aim is to provide a friendly front-end interface for users to experiment with the framework. However, in order to both focus on research and learn more new things, the development of a final UI was postponed to later releases and further work, but we prototyped a solution. A simple example of how it was prototyped can be found at Figure 4.10.

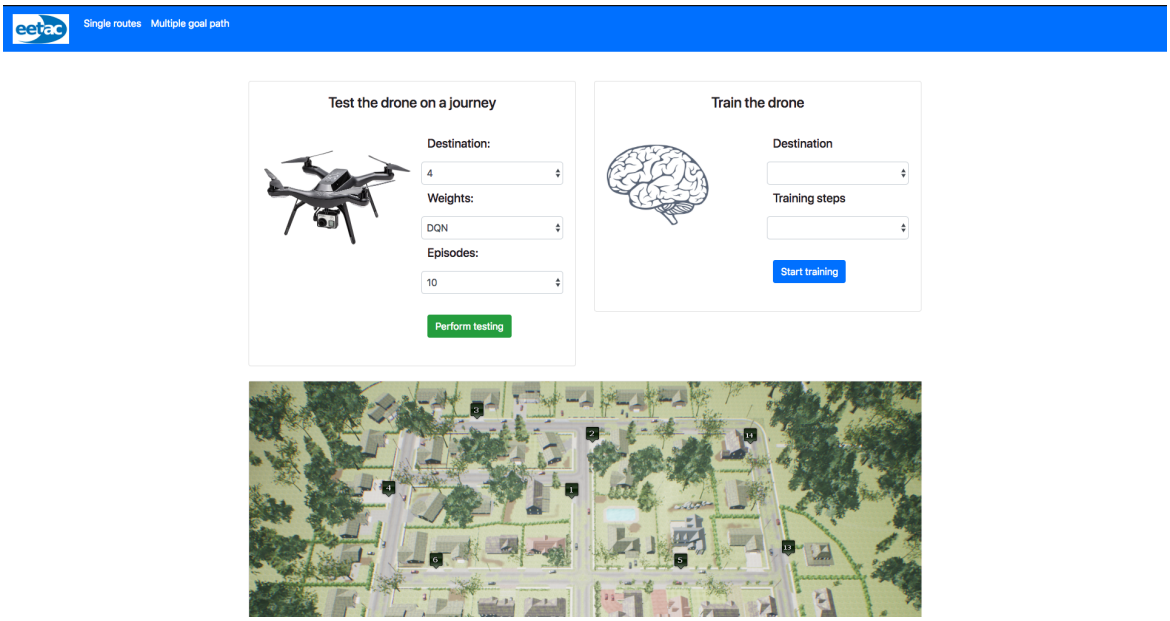


Figure 4.10: Front-end prototype built for training and testing user interface.

To provide a whole picture of how to scale this project into a production environment for research, Figure 4.11 is shown. Noticeably, the both front-end part (with improvements and tests) and the back-end side won't need huge effort to achieve a production performance, in terms of coding.

Note that front-end prototype was built using Angular, a TypeScript-based open-source front-end web application platform, connecting to the API endpoints exposed using Flask, a micro web framework written in Python. The idea to add a MongoDB, an open-source cross-platform document-oriented database, is to empower the fine-tuning research on DRL models and enable a well testing results oriented environment. Storing algorithm parameters, the reference to the whole model stored in a server and its results will help extrapolate research performances. Furthermore, storing the reference to the neural network weights managed also in a server, enables to track the behaviour of the algorithm, and to help other researchers or users to use the saved weights to perform Transfer Learning [PY+10] or just re-evaluate the exposed scores. The only thing not currently achieved at this side, due to

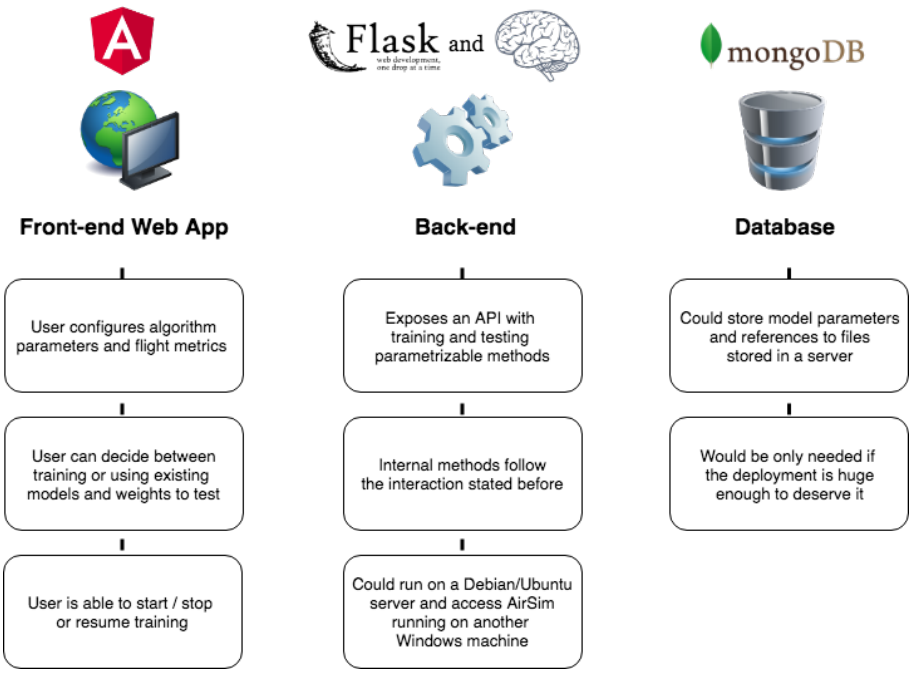


Figure 4.11: Prototype solution in a research production environment. Note that the brain icon emulates the whole deep reinforcement learning core within the Back-End, within the interaction of Gym and Keras-rl libraries.

lack of time, is running the core of the application in a Ubuntu server (as a example) and connecting to an AirSim instance on a Windows machine (AirSim binaries are only built for Windows), reproducing a drone-control station behaviour, where the control station would hold the computing power and the drone would just send the useful information.

Chapter 5

Results

This chapter is devoted to comparisons between the previously introduced architectures, training and testing performance and the analysis of the obtained results. The outline of this final chapter includes the explanation of the algorithm set-up, baseline CNN versus JNN results and different JNN modification results, together with providing useful insights for future related work.

5.1 Set-up

The training process is objectively equal in terms of the algorithm used for training. Both models are developed in Keras, hold a replay buffer (experienced replay) of 100.000 transitions and follow a linear annealed policy from a maximum value of $\varepsilon = 1.0$ to a minimum value of $\varepsilon = 0.1$ until the final exploration step, which will be 50.000 or 100.000. More details about the hyperparameters chosen are displayed below in Table 5.1.

The whole training phase is up to 125.000 steps or 250.000 steps, which stand for around 42 or 84 hours respectively on a NVIDIA GeForce GTX 1060 and 16GB of RAM. The agent optimization is achieved using an Adam optimizer, which meant a huge improvement in the way neural networks are trained and tested [KB14].

Table 5.1: Hyperparameters of the agent implementation for both models

Hyperparameter	Value	Description
training steps	125 000 or 250 000	Total number of agent-environment interactions.
minibatch size	32	Number of experiences which each stochastic gradient decent step (SGD) update is computed.
replay buffer size	100 000	SGD updates are randomly sampled from this number of most recent experiences.
target factor τ	1×10^{-2}	Factor which defines the soft update from actual network to target network.
discount factor γ	0.99	Discount factor used in the Q-Learning updates to discount the estimated return.
learning rate α	2.5×10^{-4}	The learning rate used by Adam optimizer.
initial exploration ε	1	Initial value of ε in the behaviour policy ε-greedy.
final exploration ε	0.1	Final value of ε in the behaviour policy ε-greedy.
final exploration step	50 000 or 100 000	The number of steps over which the initial value of ε is annealed to its final value.

5.2 JNN vs CNN

This section compares the results of the baseline architecture (which is extensively explained in Appendix B), which takes just the image as input (CNN model) and the second one (JNN model) which takes, as introduced before in 4.4.1, a concatenation of image, position, distance to goal and geofencing information. The section will later be splitted into training metrics and testing results.

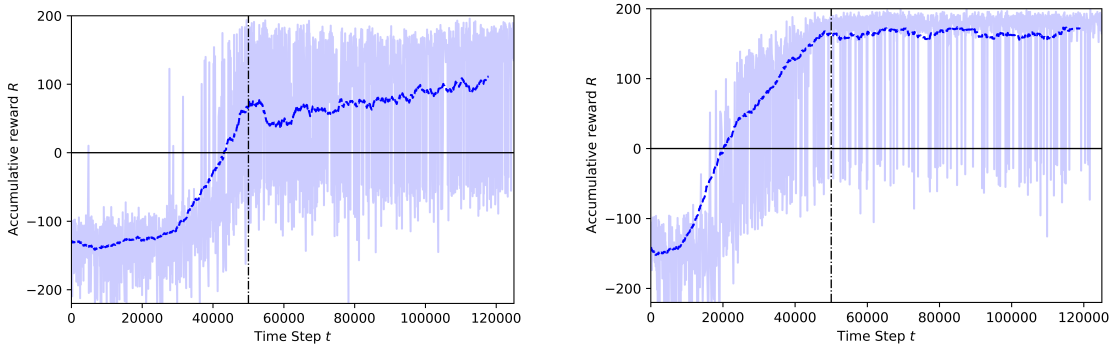
The intention of this comparison is basically to evaluate if two different approaches for the same task can show meaningful results and how the agent can adapt in a more optimal way to the environment and the goal.

The task consists in going from a starting point $x = 0$ and $y = 0$ to a goal location ($g_x = 137$ and $g_y = -48$) at the yard of a house, and land there. This task intends to simulate a real delivery process with the ability to navigate autonomously.

5.2.1 Training metrics

In Figure 5.1, a clear example of the difference in convergence between both models is shown. The JNN approach is able to converge faster over the annealed part and stays at a similar performance during the whole remaining steps, with a 10% random *behavior policy*, achieving a mean reward over 175.

Against it, 125.000 steps are not sufficient for the CNN approach to converge into a solid policy. It is less efficient during the annealed part and therefore it is able to improve during the remaining steps but it is only capable of achieving a mean reward around 100.



(a) Values obtained for the CNN model

(b) Values obtained for the JNN model

Figure 5.1: Values regarding cumulative reward obtained during training phase. Vertical discontinuous line represents until which number of steps ε is annealed to its final value.

5.2.2 Testing results

In this subsection we will evaluate if the assumptions made regarding to the training phase are translated into a level of learning capable of understanding the task. For the testing part, the same task is evaluated taking into account that the environment is not deterministic. The simulator has its own effects that vary time to time, such as for example the effect of winds on trees. Also, online computation can lead to a possible misleading prediction if the timings are not correct.

The result of a 100 episode testing for each model, with and without checkpoints is shown below.

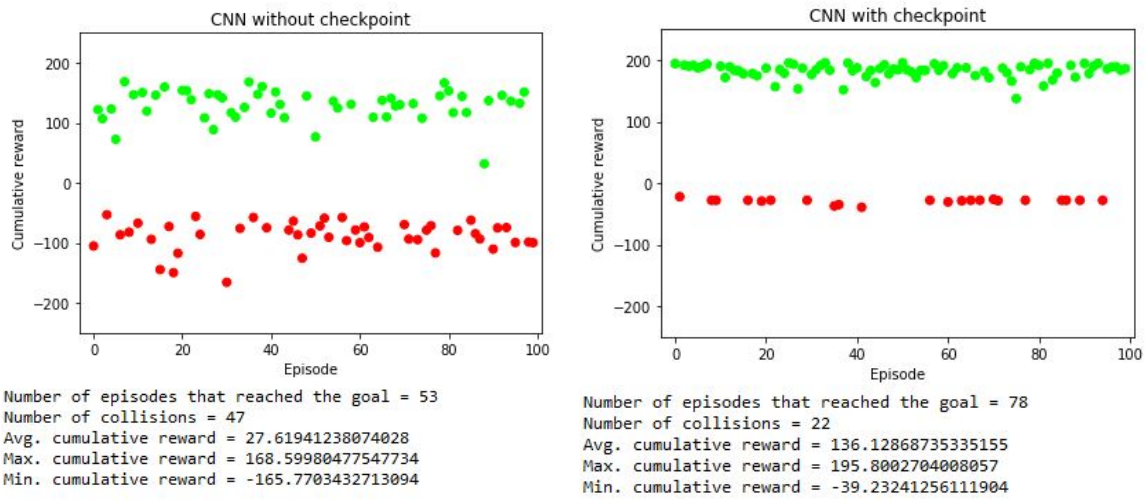


Figure 5.2: Testing episodes using the CNN model. Green dots refer to goal achieved episodes and the opposite for red ones.

Noticeably, as it is exposed in Figure 5.2, the ability to save the weights after the best performing episode is useful when training phase is not enough to provide with solid knowledge of the environment. A huge improvement is made in terms of both reward values and number of goal achieved terminated episodes. Although in 5.3 the difference is not so broad and even the maximum reward is better at some episode during testing for the JNN model without checkpoints, they still have some impact regarding the average reward, providing more smooth and identical episodes.

The behavior deduced from the training phase performance is fully supported by testing values. The JNN model adapts extremely well to the environment, providing a more efficient approach in order to required convergence time for delivering impressive results, in comparison with previous baseline architectures, drawn out from Figure 5.2. Having obtained this performance in test is unbeatable for us, as humans, since the agent achieves the goal with the JNN in the minimum steps possible within the environment and actions defined.

With this milestone achieved, the only way to go further is provide the agent with the sufficient information for him to learn to extrapolate different goals, including ones it has never reached during its training phase. In order to address that, and being demonstrated that a Double DQN agent is able to learn from both image and separate joint values, the architecture will be scaled to an upgrade, making it closer to a real-world application.

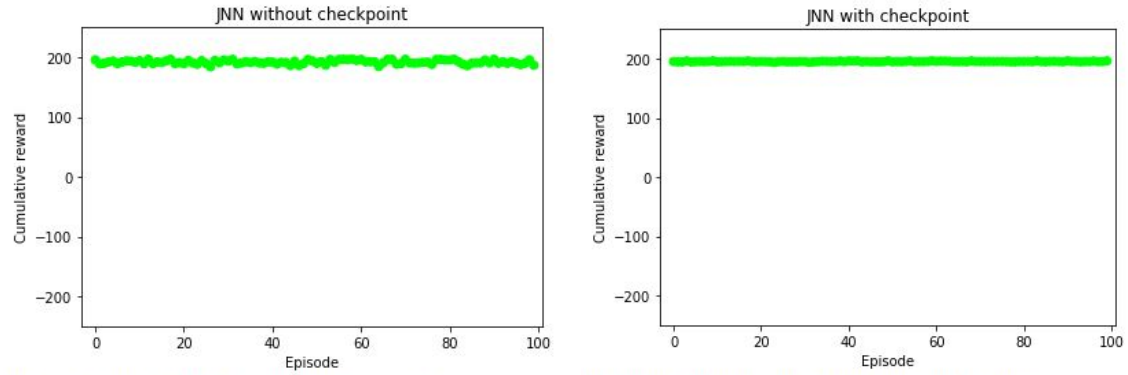


Figure 5.3: Testing episodes using the JNN model. Green dots refer to goal achieved episodes and the opposite for red ones.

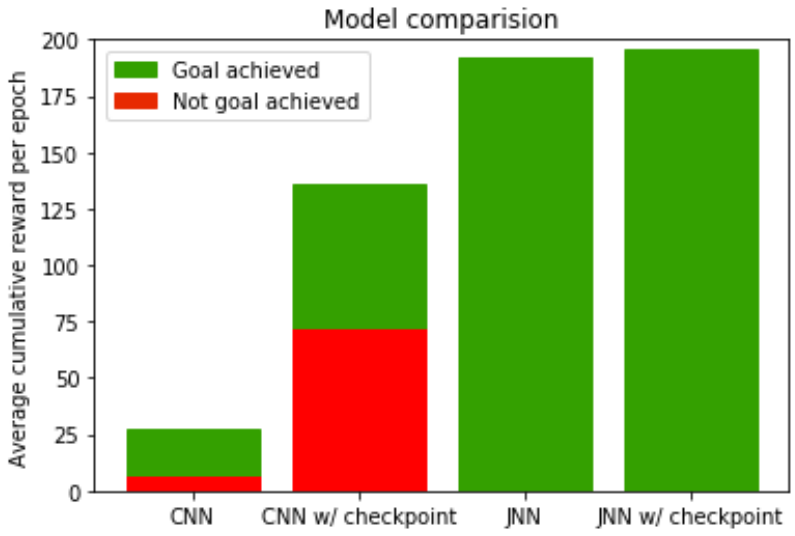


Figure 5.4: Test comparison between models.

5.3 Performance of JNN variations

This section shows results and analyzes how the modifications and improvements previously introduced affected the behavior of the agent. In order to be transparent with the following results, clicking in this *link* allows anyone to download the final and checkpoint weights of every modification of the architecture, together with all the training and testing graphs. All graphs are obtained with *matplotlib* and *pandas* libraries. This encourages the reader to test the weights with the code available on the Github *repository* and analyze its results.

The hyperparameters used for the agent remained equal to the ones used in section 5.1, except for only two values. The final exploration step was changed to 100.000 and the training steps to 250.000. However, in one case, the training steps were again reduced to 125.000 influenced by its good performance.

In order to better understand and comment over the results in a orderly manner, the following abbreviations are used: *V* stands for the architecture developed at Subsection 4.4.2 and *T* stands for the same plus the modifications explained at Subsection 4.4.3. The number attached refers to the usage or non usage of the environment exploration concept explained at the part of Multiple destinations inside Subsection 4.3.1. If the value is 1, then the task is the same (i.e. the destination never changes) and if the values is 4, the task is different (i.e. there is a random change in destination coordinates). Additionally, *Last* stands for the weights saved at the end of the training phase, and *Best* stands for the checkpoint weights saved at the end of the episode with the highest reward.

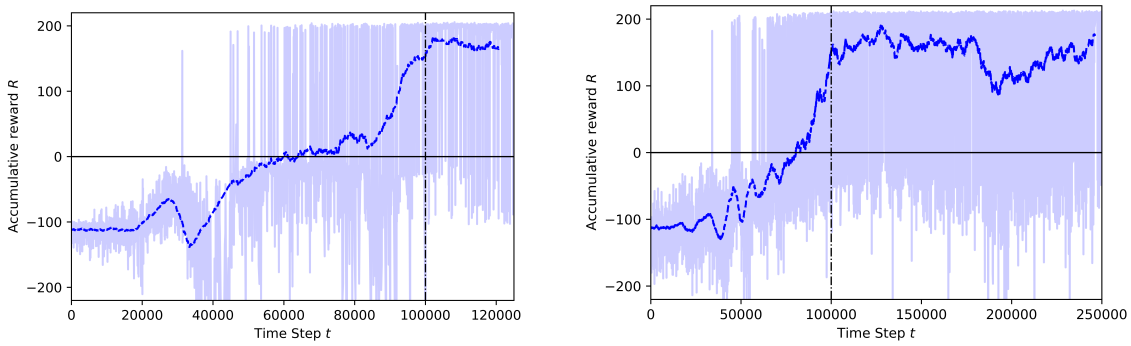
5.3.1 Training metrics

The specific values of the training are shown at Table 5.2. As described before, V1 approach was the one that performed better enough in testing mode to be forced to reduce to the half of the training steps, which is directly proportional to reducing its training time also to its half.

Table 5.2: Different training values from modifications of the general JNN architecture.

	Steps	**Time**	**Max reward**
V1	125.000	42 hours	205.57
T1	250.000	84 hours	212.69
V4	250.000	84 hours	202.63
T4	250.000	84 hours	203.23

Figure 5.5 shows how V1 architecture training convergence is nearly the same as T1 with half of the training time. Against it, T1 achieves a higher maximum reward, and therefore a better checkpoint.



(a) V1 architecture. **(b)** T1 architecture.

Figure 5.5: Training graphs comparison between the V1 and the T1 architectures.

However, both architectures show at Figure 5.6 how they are not capable to obtain such amazing values when facing more than just one destination. V4 and T4 nearly achieve the same maximum reward during training, and their annealed phase does not provide enough flexibility for the agent to perform *greedy* enough. As the graph suggests, the need for more computational time is required in order to perform at a level of the other approaches.

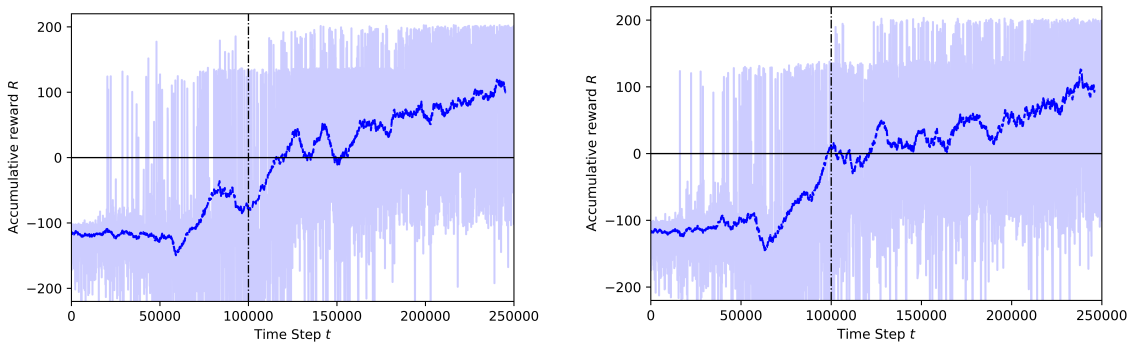


Figure 5.6: Training graphs comparison between V4 and T4 scenarios.

Figure 5.7 provides additional information for the V1 architecture, to serve as an example of the training performance. During this phase, the number of steps per episode increases near the extremes of the central zone. This means three things, UAV crashes are more frequently

at the beginning, the agent stabilizes during the central period of the training, and at the end of the phase, the UAV performs the task good enough to solve it in a few steps. On the other hand, the mean Q-Values, as expected, remain similar to the accumulative reward.

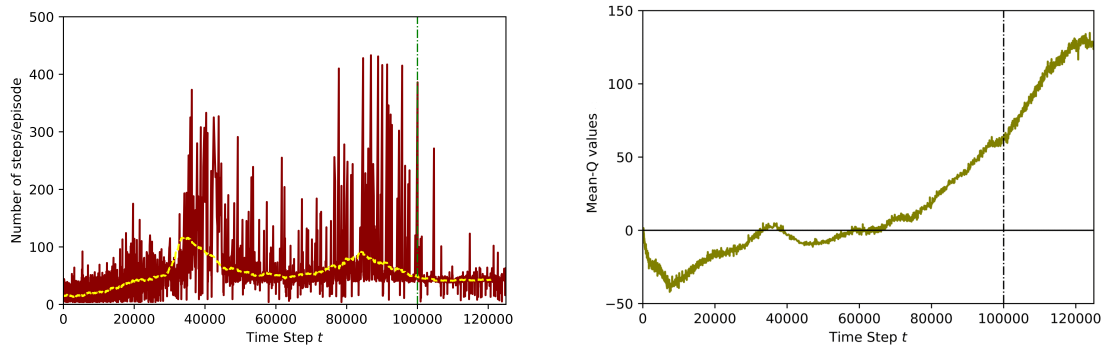


Figure 5.7: Additional training graphs for the V1 architecture.

5.3.2 Testing results

The following results show the agent facing 100 episodes within the same goals as the encountered in the training phase with either the weights saved at the end (Final) of the training phase, or the ones saved at the best checkpoint. The most interesting values are expressed by **Landed**, which basically tells if the agent has accomplished the task, and the **averaged reward**.

Analyzing results in Table 5.3, both V1 and T1 with Best weights outperform the final ones. In the case of the V architecture, the difference is minimal or nearly zero, but for the T architecture it means a huge change. In all the cases, except for the Final T1 results, this improvements beat the first obtained test results with the JNN architecture and within the same task (same destination coordinates), available at Subsection 5.2. Note also that the new architecture makes it less biased and overfitted to specific training values.

Table 5.3: Testing results for the two one-destination different architectures.

	V1		**T1**	
	Final	**Best**	**Final**	**Best**
Landed	100	100	97	100
Avg reward	202.85	202.88	199.36	203.17
Max reward	204.73	204.75	212.74	205.62
Min reward	199.86	200.56	-61.78	200.52

The testing results of the four random destinations (see Table 5.4) display an interesting phenomena. Training the algorithm for a more general case is clearly more difficult and performs particularly worse. For T4, training time is evidently not sufficient, as the training graph also suggested. However, in the case of V4, values are not so unacceptable for a real case problem. In both cases the use of Best weights is more noticeably than in past cases and therefore necessary. Empirically speaking, it would be enough in the case of V4 for achieving the task, because it has less collisions although it usually lands with a lower reward, and therefore a low pace. To be clear, the drawback of the V model is the requirement of longer training times. Nevertheless, we only consider it good enough if there are zero collisions, due to the critical it is for an UAV to crash, it would be able to arrive with no collisions,although in a regular pace (it has lower average rewards).

Table 5.4: Testing results for the two four-destination different architectures.

	V4		**T4**	
	Final	**Best**	**Final**	**Best**
Landed	93	95	61	71
Avg reward	135.85	138.19	42.85	74.96
Max reward	200.02	204.32	201.68	204.32
Min reward	-83.4	-76.64	-198.95	-200.44

To summarize, one-destination tests approach with the new architecture features clearly outperforms past results. However, the modifications of the T architecture are not always an improvement. Both tables provide a direct explanation of this case, and it is basically that when training times are extended, as in the comparison provided in Table 5.3, in which T1 has been trained twice longer what V1 has been trained. T provides additional useful information for the agent, because it actually discovers the relationship with time-sensible data. Still, training times or steps are not enough for facing multiple tasks. Additionally, T4 losses against V4 because T4 also has more additional information as input, which is not necessary for accomplishing the task (but useful for the one-task case) and confuses the agent. One could empirically say that it is nearly enough in the case of V4 for achieving the task at a regular pace, except the fact that it collides 5 or 7 times.

Besides that, it is difficult to compare the results of the four destinations with past architectures, because we actually do not have the rewards obtained when facing that destination. As the rewards vary with the destination of the goal, it is difficult to analyze detail by detail each solution. However, taking into account training times and the limited computing power, even the lowest results seem to be fairly good.

To sum up, testing results have demonstrated that JNN (Best V1) beats the baseline architecture (Best CNN) tested in 5.2.2 with the exact same conditions (same setup and goal) by achieving a 49 % more average reward (202.88 JNN vs 136.12 CNN) and finishing without collisions, against the 22 of the CNN.

An extensive and interesting video of all the performances is available clicking on these URLs from *Youtube* and *Drive*.

5.4 Looking forward

This section overviews the main proposals for improvements in the deep reinforcement learning area for UAVs, specially for our framework. All of them have been studied, and some nearly implemented. Yet it would have meant a lot more of demonstrations and comparisons for the certain objectives of this thesis, plus more training time and computational resources, that would have made the thesis too extensive.

5.4.1 Prioritized experience replay

Prioritized Experience Replay, also by DeepMind researchers, is one of the last improvements on DQN was published at the International Conference on Learning Representations (ICLR) 2016. It outperformed DQN with non-prioritized experience replay on 41 out of the 49 Atari games.

The paper [SQAS15] shows how *prioritizing* which transitions to replay can make experience replay more efficient and effective than replaying all transitions uniformly. The idea lays in that an RL agent can learn more effectively from some transitions than from others. There are transitions that may be more or less surprising, redundant, or task-relevant. Literally, *"Experience replay liberates online learning agents from processing transitions in the exact order they are experienced. Prioritized replay further liberates agents from considering transitions with the same frequency that they are experienced"*.

The researchers propose to replay transitions with high expected learning process more frequently, measured by the magnitude of their temporal-difference (TD) error. As they analyze, this prioritization lead to a loss of diversity, alleviated with stochastic prioritization, and introduced bias, which was corrected with importance sampling.

Besides the theoretical part, it has been recently released on Open AI Baselines [DHK+17] framework, a properly implementation of the Prioritized Experience Replay for DQN algorithm. This release makes easier to incorporate this improvement in our RL drone framework in a near future.

5.4.2 Transfer learning

Transfer learning is one of the most important research problems nowadays in machine learning. It primarily focuses on storing the knowledge gained while solving one problem and applying it to a different but related problem. Transfer Learning basically works when the underlying distributions are not totally unrelated.

Training can spend up to 82 hours in our project. Other DRL researches report training periods that go from days to weeks or even months. Starting a new training from scratch or randomly is not a practical solution. Recent studies [TFR+17] have worked on Transfer Learning in Deep Reinforcement Learning for vision-based control policies to be transfered from the simulator to the real world. This could be directly applied in the future work of our project.

From our point of view, Transfer Learning, besides potentially enabling us to transfer directly from simulated samples to real world, opens up a new window in tasks accomplishment. Transfer Learning theoretically allows to train an agent to learn one task, and thereafter apply this knowledge to learn another task. This is not that surprising, but when doing it all over again, and across different domains, it opens a new paradigm.

Moreover, we have applied it in our framework to test that it could be done. Keras makes practical and easy to train an algorithm for some few hours and to save the weights of the trained network and the state of the model. By keeping the optimizer and the learning rate values, which are necessary, one can later re-train the agent within another task. Nevertheless, the key point here is how to efficiently save to disk the replay buffer samples (or at least a part of it) to later keep the agent learning from previous samples, and not forget them.

Experts in Artificial Intelligence see the future with a huge database of models. Many diverse tasks will use Transfer Learning as the best enabler for reducing computation time. The amount of computation needed to learn the tasks separately will be much higher than learning all at once or transferring it.

> *"I think transfer learning is the key to general intelligence. And I think the key to doing transfer learning will be the acquisition of conceptual knowledge that is abstracted away from perceptual details of where you learned it from."*
>
> - Demis Hassabis, CEO DeepMind

Chapter 6

Conclusions

We have been able to successfully beat the existent baseline Double Deep Q-Learning architecture for autonomous UAVs [Ker18], obtaining a 49% more of average reward and no collisions, on a non-trivial task within a realistic simulated environment.

As from the beginning, the main motivation of this project has been enabling UAVs learn how to behave in a unknown environment by experience, and we are glad to conclude that we have been able to scale a solution. A practical, efficient and modular solution. This solution is available for anyone out there whose intention is to contribute to artificial intelligence for UAVs in a democratic way, supporting the open-source community.

During all the project, the environment behavior has been improved since the baseline scenario, using a neighborhood scenario with extreme realism, together with geofencing limits and landing effects when reaching the goals. Noticeably, the main contributions have been the research and architecture modification of Deep Reinforcement Learning models, and the enhancements in their efficiency training, such as checkpoints. In terms of the UAV field, we have established a scalable framework with real three dimensional control and dynamics, which brings us the possibility of transferring the research to the real world, with physical agents and environments.

Artificial General Intelligence is still not here. However, algorithmic, real-time rendering and computing limitations are still here, and there is a lot of research to be done. Future work in this area is expected to be within enabling a more general type of learning, in which Transfer Learning is the trend.

We finally wish that the research carried in this project will serve as a reference for future work and will lead to more improvements of the existing architectures within Reinforcement Learning problems.

"For all the progress made, it seems like almost all important questions in Artificial Intelligence remain unanswered. Many have not even been properly asked yet."

- François Chollet, Creator of Keras

Bibliography

[B+09] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[Bos14] Nick Bostrom. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press, Inc., New York, NY, USA, 1st edition, 2014.

[C+15] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. *GitHub, GitHub repository*, 2017.

[ER12] C.M. Elson and Y.A. Rozanov. *Markov Random Fields*. Springer New York, 2012.

[FBB+17] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[GKM10] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1-4):65, 2010.

[GM18] Mikel Garralaga Guillem Muñoz. Aerialclassifier. https://github.com/guillem74/AerialClassifier, 2018.

[Has10] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[Hum15] Todd Humphreys. Todd humphreys: Don't overregulate drones. *Alcalde*, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[JMS18] Nathaniel Rochester John McCarthy, Marvin Minsky and Claude Shannon. Dartmouth workshop, 2018.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Ker18] Kjell Kersandt. Deep reinforcement learning as control method for autonomous uavs. Master's thesis, Universitat Politècnica de Catalunya, 2018.

[KG13] Brian Karis and Epic Games. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, 2013.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[L+15] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, page 20, 2015.

[LB+95] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[LHP+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[Lin93] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[LVV12] Frank L Lewis, Draguna Vrabie, and Kyriakos G Vamvoudakis. Reinforcement learning and feedback control: Using natural decision methods to design optimal adaptive controllers. *IEEE Control Systems*, 32(6):76–105, 2012.

[MBM+16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[MMO95] James L McClelland, Bruce L McNaughton, and Randall C O'reilly. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102(3):419, 1995.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[Nie15] Michael A Nielsen. *Neural networks and deep learning*. Determination Press, 2015.

[OPBDC10] Joseph O'Neill, Barty Pleydell-Bouverie, David Dupret, and Jozsef Csicsvari. Play it again: reactivation of waking experience and memory. *Trends in neurosciences*, 33(5):220–229, 2010.

[Par12] AR Parrot. Drone 2.0. *Especificaciones Técnicas".[Online] Disponible en: http://ardrone-2. es/especificaciones-ar-drone-2*, 2012.

[Pla16] Matthias Plappert. keras-rl. https://github.com/matthiasplappert/keras-rl, 2016.

[PY+10] Sinno Jialin Pan, Qiang Yang, et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[RKT18] Eliezer Yudkowsky Ray Kurzweil and Peter Thiel. Singularity summit, 2018.

[RM85] Herbert Robbins and Sutton Monro. A stochastic approximation method. In *Herbert Robbins Selected Papers*, pages 102–109. Springer, 1985.

[Ros61] Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, CORNELL AERONAUTICAL LAB INC BUFFALO NY, 1961.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[Sch15a] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[Sch15b] Charles Schumer. Schumer proposes law. *United States Senator for New York*, 2015.

[SDLK17] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. *arXiv preprint arXiv:1705.05065*, 2017.

[Ser09] R. Serfozo. *Basics of Applied Stochastic Processes*. Probability and Its Applications. Springer Berlin Heidelberg, 2009.

[SEZ+13] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[Sut88] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[TFR+17] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 23–30. IEEE, 2017.

[TP11] Michel Tokic and Günther Palm. Value-difference based exploration: adaptive control between epsilon-greedy and softmax. In *Annual Conference on Artificial Intelligence*, pages 335–346. Springer, 2011.

[TVR97] John N Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997.

[VDW78] J Van Der Wal. Discounted markov games: generalized policy iteration method. *Journal of Optimization Theory and Applications*, 25(1):125–138, 1978.

[VHGS16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[Vin93] Vernor Vinge. The coming technological singularity: How to survive in the post-human era. 1993.

[VM] Joannes Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *ECML*, volume 3720, pages 437–448. Springer.

[VV14] Kimon P Valavanis and George J Vachtsevanos. *Handbook of unmanned aerial vehicles*. Springer Publishing Company, Incorporated, 2014.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4), 1992.

[Wer89] Paul J Werbos. Neural networks for control and system identification. In *Decision and Control, 1989., Proceedings of the 28th IEEE Conference on*, pages 260–265. IEEE, 1989.

Appendix A

Deep learning

A.1 Introduction

Deep learning is a subfield of machine learning, inside artificial intelligence, based on learning data representations, unlike task-specific algorithms suitable for supervised, unsupervised and reinforcement learning problems. For many machine learning algorithms like regression, support vector machines (SVMs), decision trees or boosting, we have an input and an output layer, and the inputs need to transformed with manual feature engineering. In deep learning, between the input and the output layers, there is one or typically more hidden layers, whose behaviour will be explained later on. Most modern deep learning models are based on an artificial neural network, although they can also include propositional formulas or latent variables organized layer-wise in deep generative models such as the nodes in Deep Belief Networks and Deep Boltzmann Machines [B$^+$09].

This scientific field has been known under a variety of names and has seen a long history of research, experiencing alternatively waves of excitement and periods of oblivion [Sch15a].

Early works on Deep Learning, or *Cybernetics* as it was used to be called back then, have been made in 1940-1960s, describing biologically inspired models such as the Perceptron, Adaline, or Multi Layer Perceptron [Ros61]. Afterwards, a wave called *Connectionism* came in the 1960-1980s with the invention of backpropagation [RHW86], the algorithm of choice that has persisted till the current days.

One of the most notable contributions during the 1990s was the Convolutional Neural Network (CNN), designed at the time to recognize simple visual patterns such as handwritten characters [LB$^+$95].

As experts state, the modern era of Deep Learning started in 2006 with the creation of more complex architectures. Regarding to computer vision and CNNs, the most important releases have been made along with the scientific competition named ILSVRC. This progress

is undoubtedly due to the Big Data era that we are experiencing right now, large sets of images of any kind and computing capabilities are more available than ever before, thus more focus on research is implicitly given. However, CNNs still posse inherent limitations. From a theoretical perspective, Deep Neural Networks are not well understood due to their non convex property. Despite numerous efforts, a proof of convergence to good global minimum has never been found. From a practical perspective, their need for large amounts of training samples does not provide them the ability to generalize when trained on small and medium datasets. However, when trained on enormous dataset, they perform extremely well.

A.2 Convolutional neural networks

Convolutional neural networks, typically named ConvNets or CNNs, are a class of deep, feed-forward artificial neural networks. Inspired by the visual cortex, they are explicitly designed for using images as inputs, being excellent for detecting local patterns in images.

CNNs use relatively little pre-processing compared to other image classification algorithms. Learning the filters that in traditional algorithms were hand-engineered is one of its major advantages, being independent from previous knowledge.

Deep neural networks, consist of an input and an output layer, as well as multiple hidden layers. Typically, their hidden layers are linear and activation layers. However, for CNNs, as were are going to introduce later, we have to add also convolutional and pooling layers. This section will be structured splitted in each layer definition, the way to set-up neural networks and some additional history on convolutional architectures research.

A.2.1 Fully-connected

The term *Fully Connected* or linear layer, implies that every neuron in the previous layer is connected to every neuron on the next layer. It is the traditional multilayer perceptron neural network [**?**]. The output from the convolutional and pooling layers represents high-level features of the input image. The purpose of the Fully Connected layer is to use these features for classifying the input into the various outputs. Apart from classification adding a fully-connected layer is also a computationally cheap way of learning non-linear combinations of these features. This enables the neural network to learn combinations of those features, which is extremely powerful.

A linear layer is basically a function which applies a linear transformation on a vectorial input. It multiplies the input vectors by a weight matrix adding also the existing bias, represented in A.1.

The idea of this layer is motivated by the basic computational unit of the brain called neuron. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^{14}–10^{15} synapses. Each neuron receives input signals from its dendrites and produces output signals along its axon. The linear layer is a simplification of a group of neurons having their dendrites connected to the same inputs. Afterwards, the activation function is the enabler of output real values, introducing non linearity.

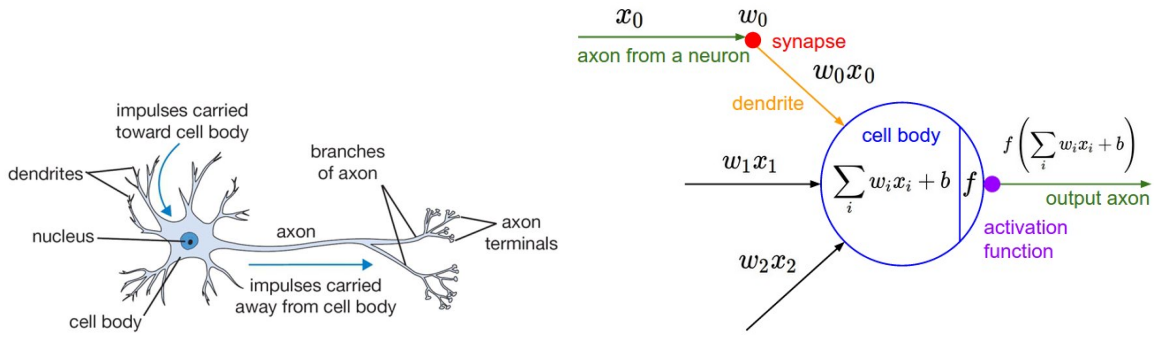


Figure A.1: Scheme of a biological neuron and its mathematical model

In a deep neural network, the last layer is a fully-connected layer followed by a *Softmax* function. The reason is that using the Softmax function as the activation function, ensures that the sum of output probabilities from the last Fully Connected layer is up to 1. The Softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that also sum up to one.

A.2.2 Activation

The capacity of neural networks to approximate any function, especially non-convex, is directly the result of the non-linear activation functions. Every kind of activation function takes a vector and performs a certain fixed point-wise operation on it. There exist many kinds of activation functions but apart from the Softmax introduced before, the three main ones for CNNs include the following:

The **sigmoid** function takes a real value and squashes it between 0 and 1. However, when the neuron's activation saturates at either extreme values of 0 or 1, the gradient at these regions is almost zero. This triggers that the back-propagation algorithm fails at modifying

its parameters and the parameters of the preceding neural layers. So, sigmoid functions are basically not optimal for back-propagation.

$$y = \sigma(x) = 1/(1+e^{-x}) \tag{A.1}$$

The **hyperbolic tangent** also squashes a real value but between -1 and 1, sharing the same sigmoid's drawback.

$$y = 2\sigma(2x) - 1 \tag{A.2}$$

On top of this, the **rectified linear** function is the strongest option at the time of writing this thesis for stacking layers in CNN architectures. Using a simple mathematical form it does not involve expensive exponentials. In the last few years it has become very popular due to its linear non-saturation form, which accelerates the convergence of stochastic gradient descent compared to the previously introduced functions in a factor of 6 [KSH12]. However, the rectifier eliminates all the negative values, which makes it not suitable for all datasets and architectures.

$$y = max(0,x) \tag{A.3}$$

A unit employing the rectifier is also called a rectified linear unit (ReLU). A smooth approximation to the rectifier is the analytic function which is called the **softplus** function.

$$y = log(1+e^{x}) \tag{A.4}$$

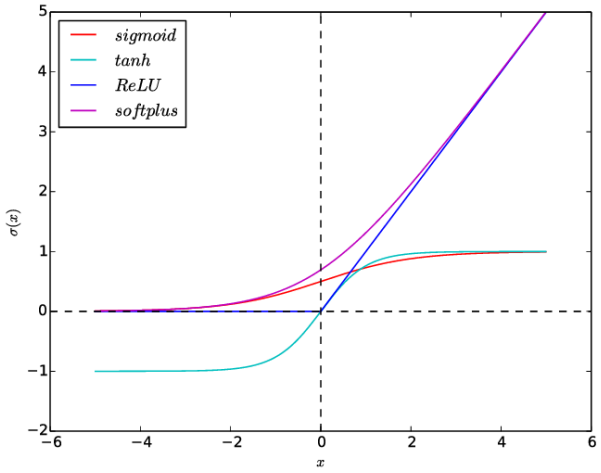


Figure A.2: A comparison of the most common non-linear activation functions

A.2.3 Spatial Convolution

Deep neural networks only made with linear and activation layers do not scale well to images. For example, for a image like 224×224 with three color channels (RGB) would need a first linear layer having at least 3 * 224 * 224 + 1 = 150,129 parameters for a single neuron. On the other hand, spatial convolution layers take advantage of the fact that their input (images or feature maps) exhibits many spatial relationships. So, a convolutional layer learns a set of N_k filters $F = f_1, ..., f_{N_k}$, which are convolved spatially with input image x, to produce a set of N_k 2D feature maps z:

$$z_k = f_k * x \tag{A.5}$$

where * is the convolution operator. In this manner, when the filter correlates well with a region of the input image, the response in the corresponding feature map location is strong. Moreover, unlike in linear layers, weights are shared over the entire image reducing the number of learning parameters, meaning that if an item is shifted in the input image will also shift the corresponding responses in a similar way.

A spatial convolution is defined by the number of filters (e.g. output channels), the properties of its filters (e.g. number of input channels, width, height) and the convolution properties (e.g. padding, stride).

A.2.4 Spatial Pooling

In order to reduce the dimension of feature maps, such as width and height and also provide invariance to slightly different input images, pooling layers are typically used in CNNs.

A pooling operation can be represented as:

$$p_R = P_{i \in R}(z_i) \tag{A.6}$$

where P is a pooling function over the region of pixels R. The spatial pooling layer is defined by its aggregation function, the width and height dimensions of the area where it is applied, and the properties of the convolution such padding and stride.

Using **Max** pooling is always the desired method due its avoidance of canceling negative elements. Pooling layer in code can be sometimes written together with the spatial convolution layer, stacking a strided convolutional layer.

A.2.5 Neural network set-up

When training CNNs from scratch, such as in other Deep Neural Networks, all the network parameters are generally **initialized** with Layer-sequential-unit-variance (LSUV), which means that all parameters are initialized as Gaussian random variables with mean 0 and standard deviation $1/n_{inputs}$ with biases also equal to zero.

In order to quantify the capacity of the network to approximate the ground truth classification for all training inputs, a **loss function** needs to be defined, which takes as inputs the weights, biases and examples from the training set. Examples of loss functions are the Mean Square Error, Mean Absolute Error or Cross Entropy. Then, the most efficient way to find the weights and biases (network parameters), taking into account the number of parameters, is to use an **optimization algorithm** such as Stochastic Gradient Descent (SGD) [RM85] or Adam [KB14] which minimizes the loss function.

For each training input, the prediction and its associated loss are computed. After that, **backpropagation** is used to propagate the error in order to compute the partial derivatives $\delta E/\delta w$ and $\delta E/\delta b$ of the cost function E for all the weights w and bias b. In order to better understand back-propagation, the second chapter of Michael Nielsen's book [Nie15] is a useful resource.

Once all the derivatives are computed, the parameters are updated using the chosen optimization algorithm as stated above. Then, the prediction (*forward pass*), the backpropagation of errors (*backward pass*) and the optimization is iterated until convergence, with the hope to find a local minimum low enough to ensure good predictions.

Hyperparameters decision, such as learning rate, weight decay, dropout usage, without mentioning, obviously, architecture decision. This is one of the most important stages in order to obtain the best performance in terms of both accuracy, loss and training time.

A.2.6 Convolutional architectures history

Although they come from long time ago, CNNs have suffered its major advances in the recent days, and a lot of convolutional architectures have been developed from the 1990's. Below, a brief inventory of the most known architectures that represented a step further in research are introduced.

LeNet-5 : [L$^+$15] Developed by Yann LeCun in the 1990's was used to read zip codes and digits. Differing on many points with previous models, it is one of the first successful application of CNNs. It introduced the way to train stacks of three layers: convolution, pooling and non-linearity, followed by fully connected layers as the final classifier.

AlexNet : [KSH12] Deeper (5 Conv, 3 Max-pool, 3 FC) and bigger (60 millions of parameters) than LeNet, was submitted to the ImageNet ILSVRC challenge of 2012 and significantly outperformed the other hand crafted models. It is the first work that made popular CNNs in modern computer vision. The authors also provided a multi-GPUs implementation in CUDA to bypass the memory needs.

Overfeat or ZFNet : [SEZ+13] Winner of ILSVRC2013 with almost 140 millions of parameters. Although based on AlexNet, the size of its middle convolutional layers was expanded, and the stride and filter size on its first layer was made smaller.

VeryDeep or VggNet : [SZ14] Runner-up architecture of ILSVRC2014 with also almost 140 million of parameters. Used to demonstrate the great advantage of that using multiple 3×3 convolution in sequence could emulate the effect of larger receptive fields, for example 5×5 and 7×7.

GoogLeNet or Inception : [SLJ+15] Winner architecture of the ILSVRC2014. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters by using average pooling.

ResNet : [HZRS16] Winner architecture of the ILSVRC2015 with 152 layers. Its main contribution was to use batch normalization and special skip connections for training deeper architectures.

Results in both 2016 and 2017 competitions, CUImage and Squeeze-and-Excitation networks respectively, were not astonishing enough to be classified as major breakthrougs in CNN research. This last contest in 2017 marked the last competition for ImageNet Challenge dataset, with the ILSVRC focusing this year (2018) in classifying 3D objects using natural language, which produces a little bit of uncertainity about how CNN research will progress together with solving their limitations.

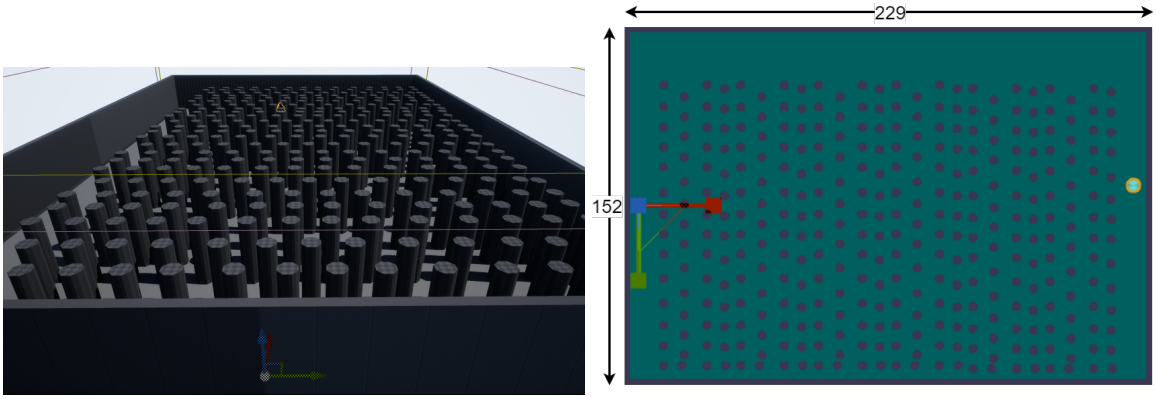
Appendix B

Previous scenario

B.1 Baseline architecture

B.1.1 The environment

The environment was built-up from Blocks package in Unreal Engine, creating a basic design with rudimentary objects, as shown in Figure B.1a. The idea was a basic rectangular space limited with walls and filled with high columns. As deduced from Figure B.1b, the initial orientation was along the positive x-axis (colored as red). The field was limited to 229 meters horizontally and 152 vertically.



(a) Three-dimensional view over the area **(b)** Bird-eye schematic perspective

Figure B.1: Unreal environment modified from Blocks. Measures are in meters

All data and variables used for the deep reinforcement learning task were presented in Table B.1. The idea was to use only data which could be obtained in a real-life scenario as well, with both internal or external sensor sources.

Table B.1: Data fetched from the simulation environment at every time-step

Data	**Meaning**
p_x, p_y	agent's global x and y position
ψ	yaw angle relative to current orientation
DepthImage	depth image in camera plan (144 x 256)
collided	boolean collision info

B.1.2 Actions

For this environment, the quadcopter was forced to move in a *xy* plane, with a fixed altitude of 6 meters. The action space consisted of three discrete actions to select at any state:

1. `straight`: Move in direction of current heading with $4\,\mathrm{m/s}$ for 1 s
2. `right yaw`: Rotate right with $30^\circ/\mathrm{s}$ for 1 s $\cong 30^\circ$
3. `left yaw`: Rotate left with $30^\circ/\mathrm{s}$ for 0.8 s $\cong 24^\circ$

The total rotation of *right yaw* and *left yaw* was slightly different in order not to define two actions that could cancel each other. Introducing both actions to rotate with 30° would had led to $360^\circ/30^\circ = 12$ possible directions throughout the environment. Combining both rotational actions, the agent had the chance to produce increments of $|30^\circ - 24^\circ| = 6^\circ$, which resulted in $360^\circ/6^\circ = 60$ possible directions to follow. So, even working just with discrete actions, the agent had a complex, realistic and non-trivial task to learn.

B.1.3 Rewards

Adapted to the task, our intention was to define an easy way for making the agent learn that we wanted to reach a certain goal without colliding in the minimum amount of time. For that reason, when a collision was detected, the episode was automatically finished and a high negative reward of -100 was given to the agent. On the other hand, when an episode was finished by reaching the goal, a high positive reward of $+100$ was given to the agent.

At this point, the intention of reaching the goal was inferred by noticing whether or not the agent was heading towards the goal. In code, it derived in being continuously analyzing if the distance to the goal after taking an action was lesser or greater than before taking it. For reaching it as fast as possible, a reward of -1 was given for any other step than the terminal ones, making the agent to minimize the amount of total steps in achieving the goal. Adding up these two concepts, for every step not evolving collision or episode ending, the reward function was traduced into $reward = -1 + (distancetogoalbefore - distancetogoalnow)$

Note that *right yaw* and *left yaw* actions did not changed agent's distance to goal, but both were -1 rewarded, motivating the agent to optimize its usage in order to maximize the reward. Moreover, take into account that distance to goal was computed as the Euclidean distance between goal's x and y coordinates and the starting one's.

B.1.4 States

Defining the state has been a crucial decision in building up reinforcement learning problems. It has to provide the sufficient information to empower the agent to learn how to reach the goal.

In this first approach, the agent received a preprocessed depth image from the front camera of the quadcopter mapped to grey scale with 144×256 pixels. Since the quadcopter moved on a fixed xy plane, only the middle section of the image turned to be relevant. So, to lighten the demand regarding computation and memory, the input dimensionality was reduced to 20×100.

Additionally, information about the goal relative to the quadcopter's position was required. The required values were goal's position, agent's position, yaw angle relative to initial orientation and the agent's relative heading to the goal, ϕ, defined as *track* in [Ker18].

$$\phi = \text{atan2}(g_y - p_y, g_x - p_x) - \psi. \tag{B.1}$$

The track angle was exposed up as the most aggregated measure of the agent's status with respect to the goal. Nonetheless, learning about it clearly shows a dependence to solve a certain task inside an environment.

This track angle information was encoded into an additional 10×100 white area as a vertical thick black line of 10×3. This tied up a 30×100 size array that represented the agent's state, as shown in B.2.

B.1.5 The neural network

Layers composing the network were stated as in [MKS+15]. The first layer was a 32 kernel 4 x 4 convolutional layer with stride 4 and ReLU activation function. After that, a convolutional 64 kernel 3 x 3 layer with stride 2 and a 64 kernel 1 x 1 layer with stride 1 both followed by ReLU activation functions were set. The final layer before the output consisted in a dense (fully-connected) layer with 512 rectified units. Ultimately, the output layer was comprised of a dense (fully-connected) linear layer with an output of the action value for each action defined in the system.

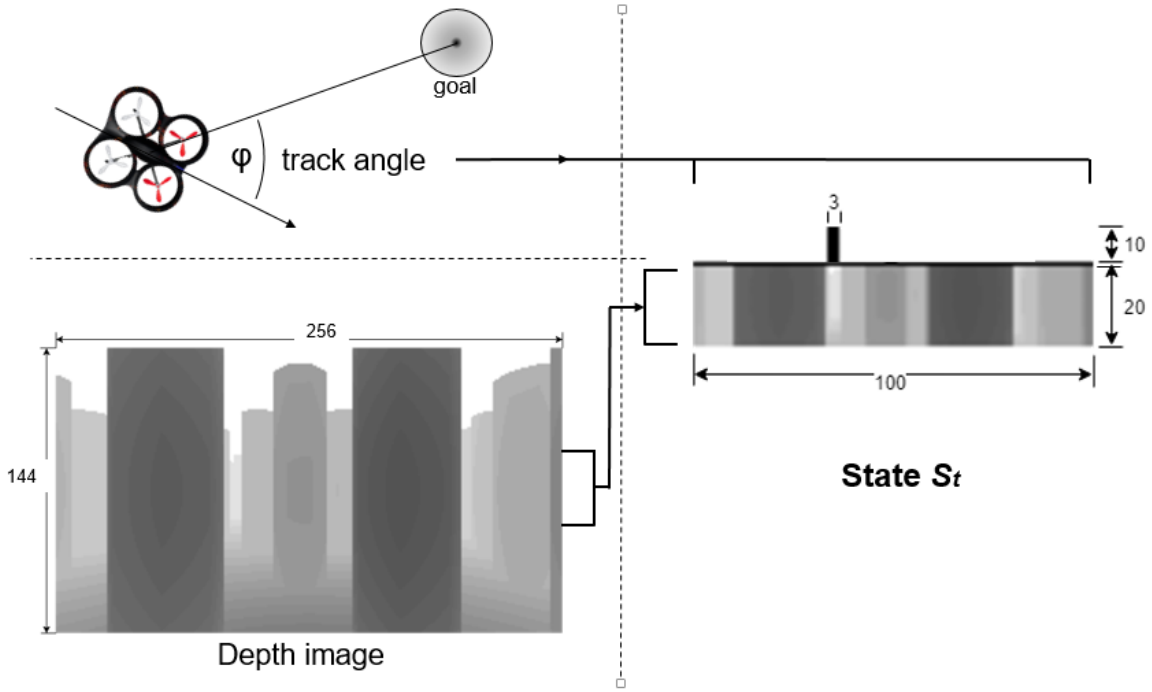


Figure B.2: State representation composed of depth image and encoded track angle. Measures are in pixels.

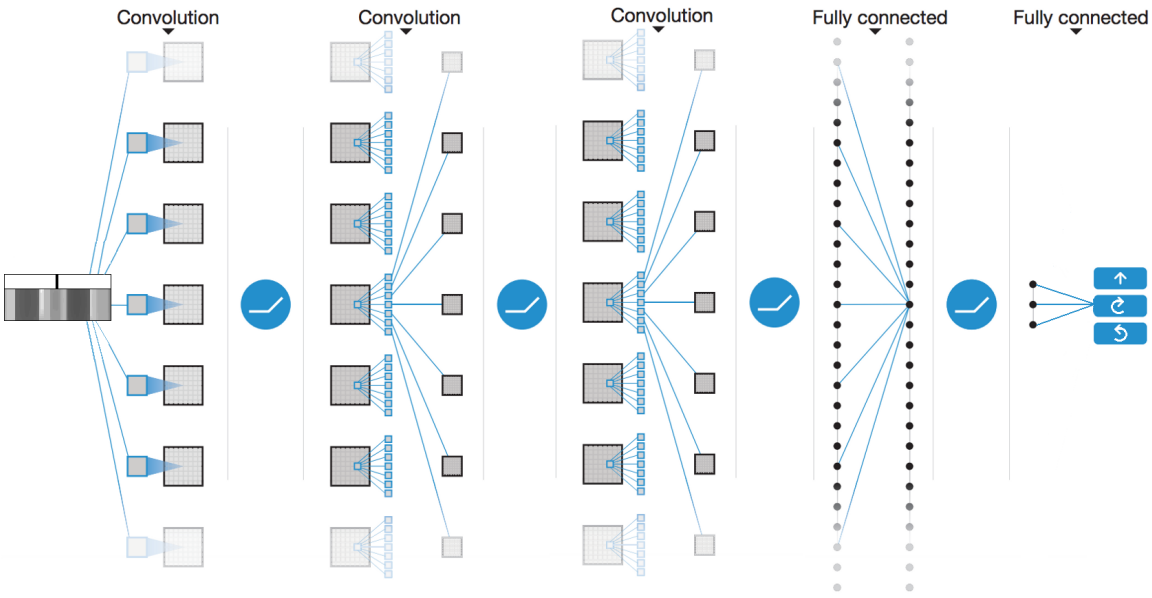


Figure B.3: Architecture of the convolutional neural network. Note that the third convolutional layer was found not to contribute to abstraction or decrease in parameter size with a kernel of 1 x 1 and stride of 1 too, but was kept to preserve the original amount of layers of [MKS+15].